A LOGIC-BASED APPROACH TO FEDERATED DATABASES

BY

WHAN-KYU WHANG

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1992

ACKNOWLEDGEMENTS

I extend a special acknowledgement to Professor Shamkant B. Navathe for his guidance and friendship during the period of my graduate study at the University of Florida, particularly through the work of dissertation. I am grateful for his boundless amount of patience, firm direction, and encouragement. His constant guidance and care will never be forgotten, even while he has been at the Georgia Institute of Technology during the last year.

I wish to express my sincere appreciation and gratitude to Professor Sharma Chakravarthy for his invaluable criticisms, stimulating discussions and helpful suggestions, through which many concepts are clarified and realized.

I wish to thank Professor Herman Lam for his time and interest in my research as well as for being my co-chairman. I am especially grateful for his invaluable comments on the dissertation. I would also like to thank Professors John Staudhammer and Manuel Bermudez for their time in the work and for serving as members of the committee.

This research was made possible by my involvement with the FIB (Federated Information Bases) project, headed by Professor Navathe and supported by the U.S. West Advanced

Technologies. Thanks go to Professor Navathe for giving me the opportunity to research the alternative approach to the project presented in this dissertation. Thanks are also due to all the members involved in the project.

I thank the University of Florida Database Systems Research and Development Center for the environment of research and access to its facilities. Special thanks are given to Sharon Grant for her friendly cooperation and support.

I would like to acknowledge my late father Who-Keun Whang, my mother, and parents-in-law for their encouragement and support. Finally, to my wife Ok-Ran and my daughter Soo-Jin I express thanks for their patience and sacrifice over the years.

TABLE OF CONTENTS

Abstract of Dissertation Presented to the Graduate School of the University of Florida in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

A LOGIC-BASED APPROACH TO FEDERATED DATABASES

By

Whan-Kyu Whang

May 1992

Chairman: Dr. Shamkant B. Navathe
Major Department: Electrical Engineering

Current systems for federated database management have concentrated on either schema integration using semantic models or query processing by the relational model, but not both issues in a single framework. In this dissertation we explored an alternative approach to database integration based on using Horn-clause logic as a canonical, intermediate form. First-order logic is used as a language for specifying an integration of component schemas as well as expressing integrity constraints associated with global and component schemas. A first-order logic-based language is rich enough to be used for canonical representation at the intermediate level. Global to local query mapping is done using unification in logic programming. Query optimization is

performed using integrity constraints such as functional, inclusion and data integration dependencies. Moreover, the logic programming language PROLOG can provide a tool for implementation. Therefore, the logic-based approach increases processing capability by providing a single formal framework for design and implementation of integrating heterogeneous databases.

In this dissertation we deal with schemas of relational databases as component schemas. To generate a single integrated schema from component schemas at the initial phase of schema integration, five types of equivalences (EQUAL, CONTAIN, CONTAINED_IN, OVERLAP, DISJOINT) are to be identified between a pair of relations. An algorithm is developed to aid the designer in deriving new equivalences from partially known equivalences as well as to check the consistency of equivalences that are previously specified.

To provide set operations on the component relations related to one another by the five types of equivalences, we have developed an extended SQL (ESQL). The set operations in ESQL are expressed nonprocedurally in a single statement, unlike conventional SQL which requires blocks of code with set operations among the blocks.

The overall approach to processing of queries in ESQL, their optimization, and decomposition against component databases is discussed. Several parts of the proposed architecture are implemented using PROLOG.

CHAPTER 1
INTRODUCTION

With the development of database management systems (DBMSs), the users are provided with the means to effectively share large amount of data in multiple files that may have overlapping and irrelevant data among them. Instead of a multitude of separate files, a database provides each application with a single, integrated view of all the data it requires. However, as organizations evolve and merge, the need for combining information stored in multiple databases grows. Applications are no longer satisfiable from a single database, but require data in multiple databases. In many respects, the current multiple databases situation in many organizations is similar to the multiple file situation before the development of database management systems.

While the benefits of integration of multiple databases are well understood, most databases in the real world are not designed in an integrated manner. Rather, many databases are designed independently for the following reasons. First, since the requirements of the various applications in different divisions of an organization are diverse, no single database management system can satisfy all these requirements. In particular, for reasons of performance and autonomy, some

important applications may require that the data be maintained in separate databases in their own ways, even though other applications require data from several databases. Second, the lack of a central database administrator in most organizations makes it difficult to produce an integrated system suitable for all constituent applications. Third, many databases were created before the benefits of data integration were well understood. Different parts of the same organization often used different software systems (DBMSs, in-home file systems, etc.) to manage their own data.

To solve the problems mentioned above, the concept of Heterogeneous Distributed Database, or Federated Database [Heim85] has evolved. Its main goal is to provide users with an illusion of an integrated database without requiring that the database be physically integrated. Several systems have been proposed as heterogeneous or multi-database systems [Brei86, Ferr83, Kaul90, Litw86, Nava89, Rusi88, Temp87]. In a federated database environment, semantically related data may be resident on various databases under diverse DBMSs and may be represented in terms of different data models. Even if the same data model is used, it is not very likely that a real world situation would be represented in the same way by different database designers. In such databases, many difficulties arise in formulating and implementing retrieval requests that require data from more than one database.

This dissertation is concerned with the use of the logic based approach to carry out all aspects of heterogeneous systems integration (e.g., schema integration, global to local query mapping, and query optimization). It is well known that logic provides not only a theoretical foundation for relational databases, but also a practical framework for formulating various database concepts in terms of logic programming. In our approach, logic is used as a formalism for integrating heterogeneous databases at the global level. Global schema and its relationships to the constituent databases are specified in terms of rules, i.e. Horn clauses. Other systems [Brei86, Land82] require a special data definition language for schema integration. However, a logic programming language such as PROLOG plays two roles, both as a declarative schema integration language and as an implementation language. Although the relational model can be supported by first-order logic in a straightforward manner, its use is not restricted to it. Using logic as the intermediate, canonical model, any higher level user specification request for retrieval or update of data can be expressed and translated into the required local target specifications. Source level queries posed in a specific query language are first converted to this formalism by a front-end translator. The query represented in terms of Horn clauses is then optimized and decomposed into subqueries that are also expressed in the form of Horn clause. The subqueries

are then translated into existing database query languages by a back-end translator. The logic query represented in a canonical form through this formalism facilitates schema integration, global to local query mapping and application of various optimization strategies. Moreover, the logic programming language PROLOG can provide a tool for implementation. Also, semantic incompatibilities that need to be captured can be represented either as exceptions or as meta-data without stepping outside the framework. Therefore, logic programming increases the processing capability by providing a single formal framework for design and implementation of integrating heterogeneous databases. Logic programming language can also be used as a rapid prototyping language before developing a full-scale FDBS that requires a large amount of time and effort.

1.1 Scope of the Work

The most ambitious requirement of heterogeneous systems is the capability of providing a view of the system which is transparent not only in terms of the data fragmentation and location, but also in terms of heterogeneity of DBMSs. However, it is very difficult to support transaction processing in heterogeneous databases without special coordination among the individual transaction managers. Since the autonomy of the individual database systems is emphasized,

update applications are assumed to be under the direct responsibility of local DBMSs. Likewise, no facility is provided to synchronize read operations across several sites. In this dissertation we will focus our attention on the problem of providing a uniform view over heterogeneous databases. Along this line, we deal with schema integration, global to local query mapping and query optimization. We do not cover all aspects of schema integration, but develop algorithms and techniques to derive the equivalence of relations so that a part of the integration process can be automated. We develop an extension of SQL that allows us to query the federated database efficiently. We illustrate how to define a global schema composed of component schemas in terms of Horn clauses. Also, we show that integrity constraints associated with global schema can be expressed in logic. We show how to simplify a query using these integrity constraints, and how to decompose the simplified global query into subqueries that are executable in local databases.

1.2 Organization of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 provides a survey of proposed systems for federated databases. We describe the current approaches and discuss to what extent they have achieved heterogeneous database access, together with their pros and cons. In Chapter 3, we justify

the logic-based approach proposed in connection with Chapter 2 and illustrate how to handle the basic issues of federated databases from the logic-based approach. In Chapter 4, we introduce schema integration methodology in general and show how to identify the relationships among relations belonging to the same cluster to be used in the global view definition. Chapter 5 introduces an extended relational data model for federated databases and extends SQL to deal with set operations between a generalized relation and its component relations. In Chapter 6, mapping from the component schemas to the global schema is represented in Horn clauses. This applies when the component schemas are integrated into a global schema using the data model in Chapter 5. Chapter 7 deals with a query processing strategy. When a query is posed in the form of ESQL, it is translated into a canonical logic query and compiled into base relations that are accessible in local schemas. After compilation, semantic query optimization (query simplification) using functional, inclusion and data integration dependencies is performed. In Chapter 8 contributions are summarized and future research topics are discussed.

CHAPTER 2
SURVEY OF RELATED WORK

The term Federated Database was coined by Heimbigner and Mcleod [Heim85] to refer to a collection of databases in which sharing is made more explicit by allowing export schemas, which define the sharable part of each local database. The systems managing them are generally called Federated Database Systems (FDBSs). Federated database was initially used by Hammer and Mcleod [Hamm79] as a federation of a loosely coupled set of components such as objects, records, and types. This notion was then extended to that of a federation of loosely coupled databases without global schema [Heim85]. A similar notion was proposed as multidatabase by Litwin and Abdellatif [Litw86]. The key concepts in the federated approach are autonomy and cooperation in interdatabase sharing. On the other hand, the multidatabase approach shares these goals, although it stresses the concepts of multidatabase manipulations and interoperability in terms of language. Another frequently used term is a heterogeneous distributed DBMS (HDDBMS). In particular, HDDBMS was intended for the integration of databases with heterogeneous data models. In practice, however, the terms federated database system, multidatabase system, interoperable database system,

and heterogeneous distributed DBMS are broadly used as synonyms to denote the integration of distributed, heterogeneous databases without distinguishing the subtleties. The three features which are characteristic of a federated database system are as follows [Shet90]:

1) Distribution: Data may be distributed among multiple databases. In a homogeneous distributed database, distribution (i.e., vertical and horizontal partitions, or replication) is deliberately conducted to increase availability, reliability and performance of data access. In a federated database, by contrast, data distribution is already present in the form of separate databases within multiple database systems before an FDBS is built. In a truly distributed DBMS, all transaction processing is done with location transparency and hence every transaction is regarded as a distributed database transaction. In a federated DBMS, such is not the case. Local transactions are autonomously handled by the local DBMS while the global federated transactions are subject to the global transaction processing (ideally with complete location transparency for the user).

2) Heterogeneity: Heterogeneity may occur in various forms ranging from hardware and software (e.g., operating system) heterogeneities to differences in computer networking protocols. The important types of heterogeneities in the database context lie in the differences in such aspects as

data models, query languages, and transaction management strategies. Even though all local databases are based on the same data model and same database management system, semantic heterogeneity occurs when the meaning of the same data or related data is in disagreement with intended use.

3) Autonomy: Each of the database systems is under separate and independent control. The type and level of autonomy are different depending on various degrees of integration. Multidatabase approach, for example, allows full autonomy architecture by managing of interconnected collection of autonomous databases without global schema. There are different types of autonomies. For example, execution autonomy deals with the local DBMS having control over the execution of a subtransaction.

In general, there are two approaches in FDBSs depending upon the explicit representation of schema integration. One approach taken from multidatabase [Litw86] does not have explicit schema integration. Thus, there should be a facility allowing the user to derive an answer to a multidatabase query by explicitly manipulating several databases. The other approach proposed by most FDBSs uses global schema in terms of explicit schema integration [Bati86].

In the terminology used in Sheth and Larson [Shet90], FDBSs can be categorized as loosely coupled or tightly coupled based on who manages the federation and how the components are integrated. Loosely coupled FDBSs [Litw86] correspond to no

explicit integration, whereas tightly coupled FDBSs involve explicit integration. Most proposed systems belong to the tightly coupled FDBSs [Brei86, Chun87, Ferr83, Kaul90, Kris87, Nava89, Rusi88, Temp87]. In loosely coupled FDBSs, the user is responsible for creating and maintaining the federation without any control enforced by the federated system and its administrators. Other terms used for loosely coupled FDBSs are interoperable database system and multidatabase system. In tightly coupled FDBSs, the administrator is responsible for creating and maintaining the federation and controls the access to local databases.

Section 2.1 discusses loosely coupled FDBS and describes the characteristics of MRDSM as a proposed system for loosely coupled FDBS. Section 2.2 discusses tightly coupled FDBS and surveys several systems such as Multibase, Mermaid and ADDS. In general, we intend to describe the features of the system rather than details of system architecture. A large number of proposed systems are comprehensively described in literature [Bati86, Shet90, Thom90].

2.1 Loosely Coupled FDBS

A loosely coupled FDBS assumes that the user can access multiple databases without the benefit of a global schema. Thus, the essential component to be able to formulate queries is a multidatabase language used to manage interoperable

databases. Such a language should provide all the functions of a database language and allow for database interoperability. This architecture has the following advantages:

1) It provides the users with a variety of views of reality because actual interoperation among schemas is done in query invocation. This is desirable when the federation DBA is unable to anticipate the precise needs of users.
2) The user plays a role of a mediator when data value inconsistencies such as different ages for the same person occur. This can be a case that the DBA cannot resolve the conflicts in his own way, and thus leaves the decision to the user.
3) When a component database participating in the multidatabase is modified or deleted, or a new component database is included, the impacts of change can be minimally reduced because there is no global schema.

The drawbacks of a loosely coupled FDBS are as follows:

1) The user must be sophisticated to be able to formulate a query in which the differences such as names, structures and domains need to be resolved. In addition, if the underlying component database is based on different models, the mapping required in a query might be beyond the user's capability.
2) The user has to be aware of each component database so that no location, distribution, and replication transparencies

need be supported.

3) A global view generated by a query is pertinent to one query and regenerated whenever it is needed.

MRDSM

Multics Relational Data Store Multidatabase (MRDSM) [Litw86] has been developed by INRIA (France) to support multiple databases designed using the MRDS relational database management system of Honeywell.

The central concept of MRDSM is a multidatabase language that allows users to define and manipulate a collection of autonomous databases in a nonprocedural way. Such a language needs features not currently part of database languages. The first feature is the ability to use logical database names in queries to qualify data elements in different databases. Other features of a multidatabase language allow for autonomy while supporting cooperation between the administrators, especially over the data definition. They allow for a data definition to enter several schemas in the multidatabase rather than requiring one definition for each individual database, and they allow for the export and import of data definitions between databases. They also provide nonprocedural manipulation of data that may be replicated between databases and may differ with respect to names, structures, or values despite having the same meaning. Manipulations may be formulated in such a way that they are

reusable despite autonomous changes to the schemas of some of the databases. Since there is no centralized control over data, changes to schemas may indeed happen at any time.

2.2 Tightly Coupled FDBS

A tightly coupled FDBS provides location, replication, and distribution transparency. This is accomplished by developing a global schema that integrates multiple component schemas. The transparencies are managed by the mappings between the global schema and component schemas, and a federation user can query using a classical query language against the global schema with an illusion that he or she is accessing a single system. The pros and cons of a tightly coupled system can be reverse of those of a loosely coupled system mentioned in the previous section.

ADDS

ADDS(Amoco Distributed Database System) [Brei86] provides uniform access to preexisting heterogeneous distributed databases. The ADDS system is based on the relational data model and uses an extended relational algebraic query language. Local database schemas are mapped into multiple federated database schemas, called Composite Database (CDB) definitions. The mappings are stored in the ADDS data dictionary. The data dictionary is fully replicated at all

ADDS sites to expedite query processing. A CDB is defined for each application. Multiple applications and users may, however, share CDB definitions. Users must be authorized to access specific CDBs and relational views that are defined against the CDBs. The CDBs support the integration of the hierarchical, relational, and network data models.

Queries submitted for execution are compiled and optimized for minimal data transmission cost. Semijoins and common subquery elimination are the query optimization techniques used.

Multibase

Multibase [Land82] has been developed at Computer Corporation of America (CCA) to provide a logically integrated, retrieve-only user interface to a physically nonintegrated distributed database environment. The global schema is based on the Functional Data Model and DAPLEX is used as the query language [Ship81].

In Multibase, views describe integrations of the data described in local schemas. The view definition language supports horizontal and vertical fragmentations as well as mapping of the data contained in the individual local databases. Users can pose a query over any combination of local schemas or views. From the user's perspective, views provide complete location transparency. The view mechanism is also used to resolve data incompatibilities that frequently

arise when separately developed and maintained databases are accessed conjointly. Incompatibilities include (a) differences in naming conventions, underlying data structures, representations, or scale, (b) missing data, and (c) conflicting data values.

Multibase performs query optimization at both the global and local levels. At the global level, the system creates query execution strategies that attempt to minimize the amount of data moved between sites and to maximize the potential for parallel processing that is inherent when multiple distributed databases are accessed. At the local level, the system attempts to minimize the amount of time to retrieve data from a local DBMS by taking full advantage of the local DBMS query language, physical database organization, and fast access paths.

One peculiar aspect of Multibase is that it has an internal DBMS to process the intermediate results. This concept of having an internal DBMS simplifies the process of integration.

Mermaid

Mermaid [Temp87] has been developed at System Development Corporation to integrate relational databases that run on a network of VAX and SUN workstation computers. Different from Multibase, there is no internal DBMS to process intermediate results. The Mermaid front-end locates and integrates data

that are maintained by local DBMSs. It makes use of one of the component database management systems to process the intermediate results.

There are two parts to presenting a single database view to the user of the federated system. First, a global schema of all or parts of the component databases must be defined. Second, at run time the system needs to translate from the global schema into the form in which the data are actually stored.

The user is able to use a single query language, SQL, to access and integrate data from the different databases. The system automatically locates the data, opens connections to the backend DBMSs, issues queries to the DBMSs in the appropriate query language, and integrates the data from multiple sources. Integration may require translation of the data into a standard data type, translation of the units, combination or division of fields, union of horizontal fragments, join of vertical fragments, etc.

The designers of Mermaid have assumed that most of the commercially available databases are either relational, or at least available with relational interface. More importance is given to query optimization than to integration of different data models. An extended SDD-1 semijoin algorithm is used for query optimization, and it has been further extended to support fragmented and replicated relations. A replicated algorithm derived from distributed INGRES has been developed

and tested. Based on the assumption that CPU overheads dominate the network costs, this algorithm uses fragmented relations to maximize the amount of parallelism in operations. The Mermaid system is currently being converted into a commercial product by a company called Data Integration Inc.

CHAPTER 3
A LOGIC-BASED APPROACH TO FEDERATED DATABASE MANAGEMENT

Three of the major tasks involved in federated databases are schema integration, global to local query mapping and query optimization. In this chapter we consider those three aspects of federated databases from the viewpoint of a logic-based approach. In Section 3.1 we discuss schema integration from the viewpoint of using first-order logic as a language for expressing integration of component schemas. The user query represented in an extended SQL, which will be discussed in Chapter 5, is translated into a logic query. The use of logic queries facilitates global to local query mapping and query optimization. Section 3.2 discusses the mapping from a global query to local queries. Finally, Section 3.3 provides an overview of the optimization of queries expressed as logic programs.

3.1 Schema Integration

Proposed federated database systems thus far can be classified into two groups, depending on the data models used for schema integration. One is based on the relational model and the other is based on semantic or object oriented models.

Since semantic models provide a richer set of abstractions, they are used in designing the conceptual structure of databases. It is evident that semantic data models are appropriate for the conceptual modeling in the top-down design approach, but they give rise to a severe overhead for schema integration for the following reasons. First, in the bottom-up approach used in pre-existing, heterogeneous, distributed systems, the existing databases should be restructured for abstract representation of data. The overhead for mapping the record oriented data models onto the semantic models or vice versa is high. Even though much work on schema integration [Lars89, Motr87, Nava82, Nava86, Sava91] may be complete and elegant from a theoretical point of view, it is expensive and hard to apply to real implementations. Second, although semantic data models are expressive for specifying data definition, some query languages (e.g., DAPLEX [Ship81] used in Multibase [Land82]) are still procedural.

In our approach, first-order logic is used as a language for expressing integration of component schemas. Schema integration is defined by rules (intensional databases) in logic programming. Use of pure relational model (without using outerjoins, for example) does not allow the conceptual integration of nondisjoint data from distinct participating databases. Although relational views can define virtual relations, they are not powerful enough to deal with integration aspects of data such as generalization. To

overcome the deficiencies of generalization of data, Breitbart et al. [Brei86] proposed extended relational operations such as outerjoin. However, aggregating functions used to combine attribute values cannot be specified in terms of the outerjoin operation. Suppose a person who works in several divisions of one organization has more than one roles and thus has his salary distributed over several databases. The effect of adding all his salaries cannot be represented in terms of relational algebra. In general, relational views are restricted to a single database, but rules can support generalization of multiple databases and derivation interconnected by several rules. In addition, logic programming is used as a general purpose language, thus allowing aggregate operations.

Although we do not deal with the schema integration beyond the conventional data models as underlying component data models, the full power of logical rules is not restricted to it. It is unlikely that all systems in the federation support the same set of operations. In order to ensure that global requests are properly computed, the global query evaluator needs to compensate for functionality that may be lacking in some underlying systems. This can be achieved with a system that provides more powerful functionality than the ones participating in the federation. For example, as opposed to conventional data manipulation languages, logical rules have the power to compute transitive closures, such as

managerial hierarchies in organization. As a result, the logic-based approach has the added advantage of providing extensibility.

From the data modeling point of view, first-order logic provides a flexible and expressive representation in the specification and design of schema integration. When various databases are integrated, certain types of constraints are required to specify the relationship between a generalized entity and its constituent entities. Those constraints that cannot be captured in structural constructs are expressed as integrity constraints outside of the data model and must be defined explicitly in the associated data model implementation language. However, formal or informal semantics of data integration are treated uniformly in first-order logic without any loss of generality.

We describe in detail the use of first-order logic as expressing integration of component schemas in Chapter 6. In Chapter 6 we show how a global schema is defined in terms of component schemas using Horn clauses. We also show that integrity constraints associated with global schema can be represented in logic. These integrity constraints are used in the query optimization, which is described in Chapter 7.

3.2 Global to Local Query Mapping

The user is provided with a global, integrated view of constituent databases through schema integration. Now the problem is how to translate the queries expressed over the integrated view into queries over constituent databases. As a kind of reverse engineering problem, it can be achieved by system-provided unification in logic programming. In other words, global views are specified by rules in logic programming. The query over the global view can be interpreted in logic programming in terms of execution of a number of sublevels of rules which eventually reference the base relations. The process in which the query is modified to refer to the underlying base relations is carried out automatically in terms of unification. However, since variable bindings between inter-rule unification are done at run time, PROLOG returns only one answer tuple each time the base relation is encountered during unification and produces another answer through backtracking on demand. To avoid the difference of query computation between PROLOG and a conventional DBMS, rules are compiled into sequences of relational algebraic operations on base relations.

Once the subqueries in the form of Horn clauses are obtained, the language translator translates the subqueries into the languages understood by the underlying database management systems. In contrast with homogeneous distributed

databases, much work is required to translate the global query language into various local query languages. Current systems do not pay much attention to these problems. It is widely accepted that a functional or logic programming language is more suitable for language translation than imperative languages. They are known to be efficient and promote rapid development of language translation systems because of their powerful symbolic computation in terms of system-provided matching mechanism (unification). Here, we use the logic programming language PROLOG for translation. The source-to-source language translation system carries out translations via an intermediate representation, Horn clause. In order to fully link a new query language into the system, two translators are required: source-input to Horn clause and Horn clause to source-output. This intermediate expression is useful not only for decomposition and optimization but also for expansibility of global query. If the global query is represented in different form such as using semantic query language or graphical interface as shown in Figure 3.1, the required number of translation is linearly increased. In other words, if the number of global query languages and local query languages are N_g and N_l respectively, then the total translations required without an intermediate language would be N_g*N_l, but with an intermediate language it would be N_g+N_l.

In the literature, an implementation has been reported using PROLOG for source-to-source meta-translation system for

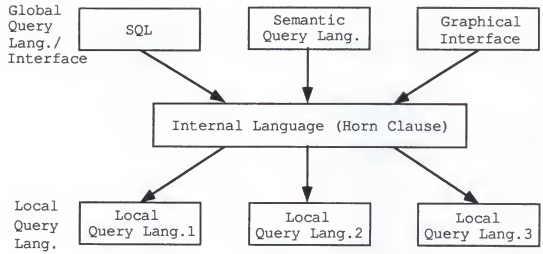

Figure 3.1 Language Translation with Internal Language

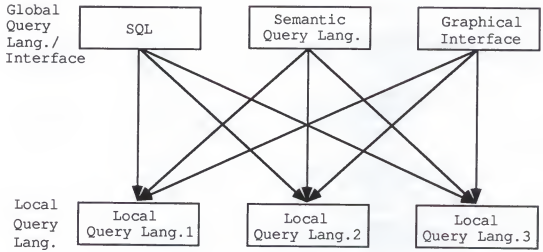

Figure 3.2 Language Translation without Internal Language

database query languages [Howe87]. They use relational algebra trees for intermediate expressions, but we believe that Horn clause is more appropriate for our purpose such as query decomposition and optimization as explained in the next section. We discuss global to local query mapping in more details in Chapter 7.

3.3 Query Optimization

It is inconvenient to optimize and decompose a high-level user query directly without any intermediate, canonical form. For this reason, many systems [Ferr83, Temp87] allow an intermediate expression for the global query in order to facilitate optimization and decomposition. Here, a logic query represented by Horn clauses serves this purpose. It provides a canonical form for the global query which can be represented in various expressions. For example, a nested query is converted to explicit join form to provide local sites with increased space for query processing strategies. The logic query constitutes a domain-oriented expression, where it is much easier to specify joins and apply optimization strategies. Now we can look at query processing associated with optimization and decomposition from the logic query viewpoint. Well-known optimization techniques such as pushing selection as early as possible can be carried out automatically during query compilation. In distributed

systems, optimization tactics such as distribution of unary operations over union, which are essential to optimization, are also applied to federated databases and automatically conducted.

A large body of work [Banc86, Ullm85] has been developed for expressly optimizing Horn clause logic programs. The intermediate, canonical representation is a natural place for performing these optimizations before translating a global query into local queries. We discuss the query optimization in more detail in Chapter 7.

In this chapter we looked into three major tasks involved in federated databases, namely, schema integration, global to local query mapping and query optimization from the viewpoint of a logic-based approach. In the next chapter we discuss a part of schema integration that is independent of the data model.

CHAPTER 4
SCHEMA INTEGRATION FOR FEDERATED DATABASES

In this chapter we discuss schema integration for federated databases. In general, schema integration, which is referred to as an integration of existing and proposed databases, occurs in two different contexts [Bati86]:

1) View integration (in logical database design): Several views of a proposed database are merged to form a conceptual schema describing the entire schema. Note that the view here refers to an application's view of the required data that is provided. It may be supported by one or more views defined over the underlying databases.
2) Database integration (in federated databases): Schemas of existing databases in use are merged into a global schema that represents a collection of these databases. The global schema can then be used as an interface to the component databases.

The actual methodologies for both types of schema integration are quite similar. The major difference lies in the way user queries and transactions are processed after integration. In the view integration, user queries and transactions specified against each view are mapped to the requests on the conceptual schema. On the other hand, in schema integration to provide

a global federated database schema, user queries against the global schema are transformed into subrequests on the underlying constituent databases. The results of subrequests are then assembled to produce a result to the original request.

Our methodology for schema integration consists of the following four steps: 1) data model conversion, 2) clustering of related entities [Nava82], 3) matching, and 4) merging. First, different data models are transformed into a common data model to facilitate the integration process. Second, the same or similar real world entities represented in different schemas are grouped together to be generalized into a generic concept. Third, the entities in the same group are matched and conflicts among them, if any, are detected and resolved. Finally, we merge these entities into a global schema. In section 4.1, we examine each step of the schema integration process in detail. In this dissertation, we do not cover all of the aspects of schema integration, which is beyond the scope of this work. Rather, we intend to develop algorithms and techniques so that a part of the integration process can be automated. Along this line, in Section 4.2 we develop an algorithm to derive the equivalences of relations whose equivalences are partially known.

4.1 Four Steps of the Schema Integration Process

The four steps of the schema integration process are illustrated in Figure 4.1 and are discussed below.

4.1.1 Data Model Conversion

In federated databases, we may have numerous local databases from heterogeneous database management systems that use different data models. Before integration, all local schemas must be converted to a common data model. Here, we use the relational model as a common data model. In this dissertation, we use the term component schema to refer to a schema that is converted to relational view from local schema. This step is not necessary for a local schema that is specified using the relational model. We assume that local databases are populated according to the relational, hierarchical and network model that are used in most existing databases. Relational view for nonrelational models such as network and hierarchical models is covered in Chapter 6.

4.1.2 Clustering of Related Entities

In order to integrate component schemas into a unified schema, we need to group the same real-world entities, which are then merged into a generalized relation. For example, the

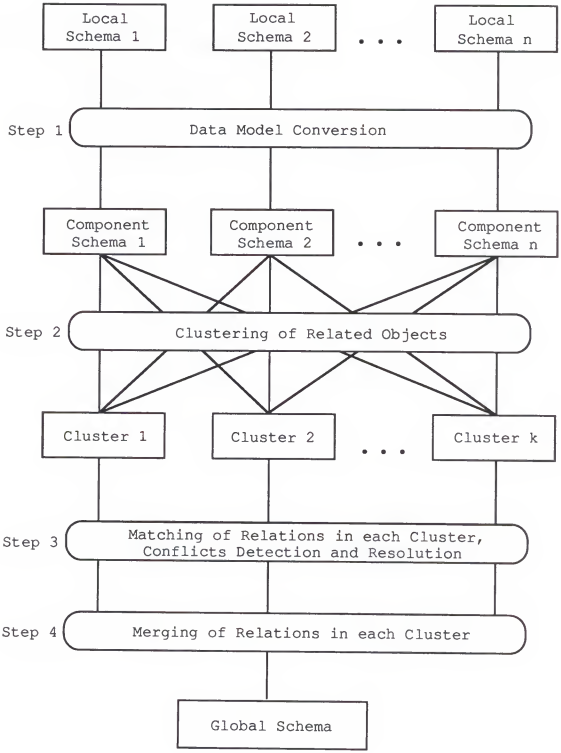

Figure 4.1 Steps of Schema Integration Process

two relations MAIN_FACULTY and BRANCH_FACULTY in the main campus and branch campus, respectively, can be merged into a general concept of relation, FACULTY. Each group is called a cluster of relations and is integrated independently of other groups. Therefore, integration problem is much simplified.

In this process, determining whether relations are grouped together is ultimately a matter of the designer's decision. For example, if one component schema has a relation SECRETARY and another component schema has a relation ENGINEER, it may be useful to integrate them as EMPLOYEE even though their real-world entities are known to be disjoint. However, that decision cannot be automatically done without referring to the entities in reality. In general, the semantics implied by the relations are not simply captured by syntactics, but the following heuristics can be used to help the designer decide clustering or to automate the process partially. First, the same names of relations are examined to determine whether they indicate the same real-world entities. When they have similar names, the data dictionaries including thesauruses are used to find the similar concepts. Second, the common key of relations can be used to broadly group the relations. For example, the two relations EMPLOYEE and PROJECT having the primary keys Soc_Sec_No and Proj_Name, respectively, cannot be clustered because they do not have a common key. Third, the number of same attributes also can be used to aid to find the same-real world entities.

4.1.3 Matching

Once the clustering of relations is done, the equivalences among relations in a cluster are to be identified. The equivalences illustrating the relationships among relations are EQUAL, CONTAINS, CONTAINED-IN, OVERLAP and DISJOINT. Determination of the type of equivalence is essential for merging of component schemas. For example, if two equivalent relations are specified in two different databases, the appropriate one is used for the global representation, and therefore minimal representation can be achieved.

After identifying the relation equivalences, the designer should then indicate the semantic equivalences of attributes of the relations in the same cluster in order to merge similar attributes in the global schema [Lars89, Mann84, Motr81]. Consider, for example, the attributes GRADE and SCORE taken from two different student databases. GRADE is defined as a one character string type, while SCORE is defined as an integer type. Although the names and data types of two attributes are different, they are semantically equivalent. The identification of this equivalence cannot be done entirely by syntax. The designer must ultimately determine the semantic equivalence of two attributes because simple syntactic rules cannot deal with such one way and two way equivalences.

During the relation and attribute matching, the designer has to detect and resolve the conflicts existing in name, structure and domain. The naming conflicts include synonyms and homonyms. The homonyms are detected by comparing concepts with the same name in different schemas and can be resolved by prefixing the homonyms by the schema name [Elma86], whereas synonyms can only be detected after an external specification such as using data dictionaries, and thesauri.

Structural conflicts arise as a result of a different choice of modeling constructs or integrity constraints. They can occur in the following four ways [Bati86]: modeling construct conflicts, dependency conflicts, key conflicts and behavioral conflicts. Modeling construct conflicts arise when the same object is represented as an entity in one schema and as an attribute in another schema. Dependency conflicts arise when the relationship between entities are represented distinctively in different schemas (e.g., one-to-one versus many-to-many). Key conflicts arise when different keys are assigned to the same concept in different schemas. Behavioral conflicts arise when different insertion/deletion policies are used in the same object in distinct schemas. For example, deleting the last employee may cause the deletion of a department itself in one schema, whereas the department may exist without employee in another.

Domain conflicts occur as a result of different physical representation (i.e., data type) for a semantically related

object. Even though the same physical representation is used for a semantically related object, they may differ in their precise semantics. Solutions for the former case might be data type conversion, whereas for the latter might be mapping from an element in one domain to a unique element in another.

4.1.4 Merging

After all local schemas are converted to relational view and conflicts mentioned above are resolved, all the component schemas are merged into a single federated database schema. We use extended relational model in Chapter 5 for schema integration. In this dissertation, we use the term global schema to refer to a schema that is being developed by integrating component schemas using the extended relational model. The component schemas themselves are in the relational model.

Among the four steps, clustering together with detection and resolution of conflicts in naming, structure and domain are not treated in this work. They appear in other literature [Bati84, Motr81], and as pointed out in Navathe et al. [Nava86], this stage is virtually an art rather than a science because much work is guided subjectively depending on the applications on hand. Below we examine the relation equivalence in terms of key attribute values of relations.

4.2 Relation Equivalence

The equivalence between two relations is determined by the values of key attributes that are in common. Let a common key exist or be defined between relation A and relation B. Without this assumption, there is no way to match common real-world entities represented in both relations. Depending upon the relationship of key attribute values, we define five types of equivalences in terms of the domain shown below. The domain of a relation is the set of tuples in that relation. Note that the equivalences should be asserted by the designer. It cannot be completely automated for the following reasons: 1) current data models cannot capture real-world state completely, and 2) the semantics of the schema may differ even in the same data model depending on the intended use.

case 1: Identical domains (EQUAL)

A EQUAL B (A = B) $\triangleq$ Dom(A) = Dom(B)

case 2: Containing domains (CONTAIN)

A CONTAIN B (A c B) $\triangleq$ Dom(A) $\supseteq$ Dom(B)

case 3: Contained domains (CONTAINED IN)

A CONTAINED IN B (A ci B) $\triangleq$ Dom(A) $\subseteq$ Dom(B)

case 4: Overlapping domains (OVERLAP)

A OVERLAP B (A o B) $\triangleq$ Dom(A) $\cap$ Dom(B) $\neq \emptyset$
$\wedge$ Dom(A) $\not\subseteq$ Dom(B)
$\wedge$ Dom(A) $\not\supseteq$ Dom(B)

case 5: Disjoint domains (DISJOINT)

A DISJOINT B (A d B) ≜ Dom(A) ∩ Dom(B) = ∅

For each pair of relations that belong to the same cluster, an assertion describing the equivalence between them must be specified to be used in the schema integration. Without the equivalence information among relations, the schema integration cannot be done because the attributes of relations have different semantics depending on the equivalences. In general, there are ${}_nC_2 = n*(n-1)/2$ equivalences for n relations. Even if the number of relations belonging to the same cluster is not so large, the number of equivalences to consider will still be large (for example, for ten relations, the number of equivalences to consider will be 45). Hence, if the number of relations to be integrated are large and only some of the assertions are specified by a designer, an algorithm is required to be able to aid the designer to derive a new assertion from the partially known assertions as well as to check the consistency of assertions that are previously specified.

A similar algorithm has been developed by Elmasri et al. [Elma86]. However, Elmasri et al. [Elma86] is not complete in the sense that only four of the five types of equivalences were taken into account. The OVERLAP case was not considered. Besides, Elmasri et al. [Elma86] does not consider all the possible cases of the transitive relationship between two

relations, but consider only the case when the transitive relationship results in a single value. In our method, we consider all five types of equivalences and transitive relationships which sometimes yield multiple values as possible options.

To help understand the designer's role for the schema integration process in Figure 4.1, we show the block diagram in Figure 4.2. In Figure 4.2 the designer identifies the equivalences among relations in a cluster. The algorithm is applied to check consistency of asserted equivalences as well as to derive new equivalences from partially known equivalences. When all the equivalences among the relations in a cluster are identified, relations in a cluster are merged into a global view relation using semantic relationships among attributes or relations from the set of equivalences. After generating a global schema, we formulate a query using the proposed extended SQL (ESQL).

The algorithm requires two inputs, namely, an Equivalence Assertion (EA) matrix and a Transitive Rule (TR) table. In the EA matrix, the equivalence assertions between relations are placed in a n by n matrix, where n is the number of relations belonging to the same cluster. The goal of the algorithm is to fill out the EA matrix with entries EA(i,j) as follows:

EA(i,j) ϵ {=, c, ci, d, o}

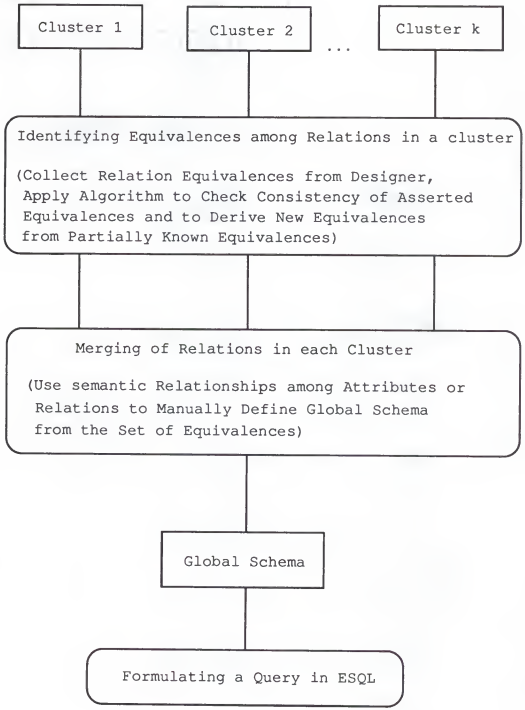

Figure 4.2 Steps for Creating a Federated Database Schema and Formulating a Query against it in ESQL

Suppose we have three databases related to faculties of a university. Database 1 (main campus) and database 2 (branch1 campus) both contain relation FACULTY, while database 3 (branch2 campus) contains three relations, FACULTY, ENGINEERING FACULTY and ELECTRICAL ENGINEERING FACULTY. Their equivalences are given as follows:

(MAIN_FACULTY OVERLAP BRANCH1_FACULTY)
(MAIN_FACULTY DISJOINT BRANCH2_FACULTY)
(BRANCH2_FACULTY CONTAIN BRANCH2_ENG_FACULTY)
(BRANCH2_ENG_FACULTY CONTAIN BRANCH2_ELEC_ENG_FACULTY)
(BRANCH1_FACULTY DISJOINT BRANCH2_ELEC_ENG_FACULTY)

These assertions capture the inter-schema as well as the intra-schema semantics specified in database 3. The graph representation of these assertions is shown in Figure 4.3. The internal representation in the form of an EA matrix is shown in Figure 4.4. In a graphical representation, a cluster is a connected graph with relations as nodes and equivalence assertions as edges. The label on the solid line in Figure 4.1 represents an explicitly stated assertion, while the dashed line represents an unspecified assertion which may be derived.

Transitive Rule (TR) table in Figure 4.5 is used to check consistency of the EA matrix and to derive correct equivalences from the EA matrix. Let E_1, E_2 and E_3 be equivalence assertion symbols between relations A and B, B and C, and A and C, respectively. E_1 and E_2 are used to index the rows and columns as shown in Figure 4.5. If E_3 is the symbol

Legend

1: branch2_elec_eng_faculty
2: branch1_faculty
3: main_faculty
4: branch2_faculty
5: branch2_eng_faculty
ci: contained in
d: disjoint
o: overlap

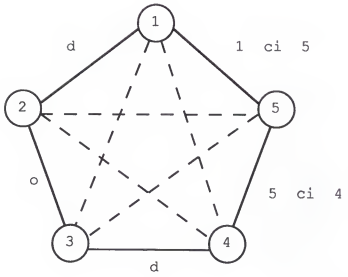

Figure 4.3 Graph Representation of Equivalences between Relations

Legend

=: equal
ci: contained in
c: contain
d: disjoint
o: overlap
u: unknown

Equivalence = (Row Equivalence Column)

Node	1	2	3	4	5
1	=	d	u	u	ci
2	d	=	o	u	u
3	u	o	=	d	u
4	u	u	d	=	c
5	c	u	u	ci	=

Figure 4.4 Equivalence Assertion (EA) matrix of Figure 4.3

Legend

=: equal
ci: contained in
c: contain
d: disjoint
o: ovelap

E_1 \ E_2	=	ci	c	d	o
=	=	ci	c	d	o
ci	ci	ci	{=,ci,c,d,o}	d	{ci,d,o}
c	c	{=,ci,c,o}	c	{c,d,o}	{c,o}
d	d	{ci,d,o}	d	{=,ci,c,d,o}	{ci,d,o}
o	o	{ci,o}	{c,d,o}	{c,d,o}	{=,ci,c,d,o}

Figure 4.5 Transitive Rule (TR) table

in the TR table, then for any three relations A, B and C the following transitive relationship holds:

(A E_1 B) and (B E_2 C) --> (A E_3 C)

Each of E1, E2 and E3 can be one of the following elements, (c,ci,c,d,o). Some of the symbols in the TR table contain more than one element, which illustrates all the possible equivalences under the corresponding transitive relationship. The algorithm uses this rule table to derive an unspecified assertion (as dashed edge in Figure 4.3). A derivation for an edge consists of two transitive edges that have only one intermediate node. Thus, the set of derivations for an edge includes all paths of length two. In general, there are n-2 derivations for any edge, where n is the number of nodes (relations) in a cluster. For example, if there are 10 relations (nodes), then there are 45 edges and for each edge there are 8 cases of transitive relationships to consider. Algorithm 4.1 is used to check the consistency of the assertions in the EA matrix previously specified by the designer and to deduce new assertions from the existing assertions in the EA matrix.

ALGORITHM 4.1: Algorithm for Checking Consistency of Specified Equivalence Assertions and Deriving a New Assertion

INPUT: Equivalence Assertion (EA) matrix and Transitive Rule (TR) table.

FUNCTION: checking consistency for the entered equivalence assertions and deriving new assertions from them.

METHOD: The first portion of the algorithm performs consistency checking for the labelled edges by building transitive paths. If the transitive relationship obtained from the TR table contradicts with the specified assertion in EA matrix, an inconsistency message is returned to the designer. For the unlabelled edges, the algorithm finds all the transitive paths for each unlabelled edge and checks the consistency for them. If the intersection of all the derived equivalence assertions on the unlabelled edge is empty, it means that inconsistency occurs. If the intersection contains only one element, the assertion is definite and inserted into the EA matrix. This equivalence information is used for next processing. This process goes on until no further derivation is possible or no further inconsistencies are detected. At the last step, the unlabelled edges whose assertions cannot be derived are returned with the possible equivalences to the designer.

```
PROCEDURE Derivation_Of_NewAssertions_And_Consistency_Checking

BEGIN
   LE := all labelled edges in EA matrix;
   NewUE := all unlabelled edges in EA matrix;

   /* check consistency for labelled edges
      building transitive path */
   T := all transitive paths in LE;
   FOR each transitive path t ϵ T DO
       IF assertion in EA is different from the transitive
             relationship in TR table
```

```
            THEN inconsistent_state;

      /* check consistency for unlabelled edges and
         deriving new equivalences using TR table */
      REPEAT
         UE := NewUE;

         FOR each unlabelled edge e_i ε UE DO
            BEGIN
               T_i := all transitive paths:
               R_i := ();

               FOR each transitive path t ε T_i DO
                  IF assertion is found in EA matrix
                     THEN BEGIN
                             r := transitive relationship
                                    in TR table;
                          R_i := R_i ∩ r
                          END
                     ELSE R_i := Unknown;

               IF R_i = ()
                  THEN inconsistent_state
                  ELSE IF R_i contains one element
                          THEN BEGIN
                                  insert the element in R_i
                                     into EA matrix;
                               NewUE := UE - e_i
                               END;
            END;

      UNTIL UE ≠ NewUE and NewUE ≠ ();

      /* for those edges that cannot be derived, a designer
         feedback is required */
      IF R_i contains more than one element or
          Ri contains unknown
         THEN output R_i's;
   END
```

Let us illustrate the algorithm using the example shown in Figures 4.3, 4.4, and 4.5. The list of the labelled edges in Figure 4.3 are [1-2, 2-3, 3-4, 4-5, 5-1]. There is no transitive paths between the labelled edges having length two. Therefore, we don't need to consider the consistency of

equivalences in the initial state, but consider the derivation of assertions of unlabelled edges. The initial list of the unlabelled edges are [1-3, 1-4, 2-4, 2-5, 3-5]. For example, the equivalence of unlabelled edge 1-3 is obtained as follows:

1) edge 1-3

transitive paths	possible equivalences
1 -d- 2 -o- 3	{ci,d,o}
1 -u- 4 -d- 3	{u}
1 -ci- 5 -u- 3	{u}

intersection of possible equivalences = {ci,d,o}

For the edge 1-3, there are three possible transitive paths: 1-2-3, 1-4-3 and 1-5-3. From Figure 4.3, the equivalences of edges 1-2, 2-3, 4-3 and 1-5, are initially given as d, o, d and ci, respectively. The edges 1-4 and 5-3 are unlabelled, therefore it is marked as u standing for unknown. When all the equivalences in the edges of transitive paths are identified, the Transitive Rule table in Figure 4.5 is used to derive the equivalence of transitive path. For example, for the transitive path 1-2-3 where the equivalences of edges 1-2 and 2-3 are d and o, the equivalences of edge 1-3 will be one of {ci,d,o} that is found in the TR table in Figure 4.5. If one of the edges is unknown in the transitive path as in 1-4-3 and 1-5-3, the equivalence in transitive path results in "unknown". "Unknown" equivalence can be taken one of the

elements, {=,ci,c,d,o}. When all the equivalences in transitive paths are identified, the intersection of them will be an equivalence of that edge. Here the intersection of equivalences {ci,d,o}, {u} and {u} results in {ci,d,o}. Likewise, the other edges are obtained as follows:

2) edge 1-4

transitive paths	possible equivalences
d u 1-----2-----4	{u}
u d 1-----3-----4	{u}
ci ci 1-----5-----4	{ci}

intersection of possible equivalences = {ci}

3) edge 2-4

transitive paths	possible equivalences
d ci 2-----1-----4	{ci,d,o}
o d 2-----3-----4	{c,d,o}
u ci 2-----5-----4	{u}

intersection of possible equivalences = {d,o}

4) edge 2-5

transitive paths	possible equivalences
d ci 2-----1-----5	{ci,d,o}
o u 2-----3-----5	{u}
u c 2-----4-----5	{u}

intersection of possible equivalences = {ci,d,o}

5) edge 3-5

transitive paths	possible equivalences
u ci 3-----1-----5	{u}
o u 3-----2-----5	{u}
d c 3-----4-----5	{d}

intersection of possible equivalences = {d}

After the first loop, the equivalences of the edges 1-4 and 3-5 are of single and definite values: {ci} and {d}, respectively. Therefore, their equivalences are inserted in the EA matrix. These equivalences are used to derive the equivalences of unlabelled edges in the next process. The second loop proceeds with the unlabelled edges, [1-3, 2-4, 2-5], as follows:

1) edge 1-3

transitive paths	possible equivalences
d o 1-----2-----3	{ci,d,o}
ci d 1-----4-----3	{d}
ci d 1-----5-----3	{d}

intersection of possible equivalences = {d}

2) edge 2-4

transitive paths	possible equivalences
d ci 2-----1-----4	{ci,d,o}
o d 2-----3-----4	{c,d,o}

```
   u     ci
2-----5-----4            {u}
```

intersection of possible equivalences = {d,o}

3) edge 2-5

transitive paths	possible equivalences
d ci 2-----1-----5	{ci,d,o}
o d 2-----3-----5	{c,d,o}
u c 2-----4-----5	{u}

intersection of possible equivalences = {d,o}

After the second loop, the equivalence of the edge 1-3 is known, while that of the edge 2-4 is the same as the first loop and that of edge 2-5 is more specifically determined. The third loop proceeds with the unlabelled edges, [2-4, 2-5], as follows:

1) edge 2-4

transitive paths	possible equivalences
d ci 2-----1-----4	{ci,d,o}
o d 2-----3-----4	{c,d,o}
u ci 2-----5-----4	{u}

intersection of possible equivalences = {d,o}

2) edge 2-5

transitive paths	possible equivalences
d ci 2-----1-----5	{ci,d,o}

```
   o     d
2-----3-----5            {c,d,o}
   u     c
2-----4-----5            {u}
```

intersection of possible equivalences = {d,o}

The third loop does not give any equivalence having a single value. Therefore, the algorithm stops here and returns the result to the designer. In conclusion, the equivalences obtained from the algorithm are given below and the corresponding EA matrix is depicted in Figure 4.6. The first three cases give single possible value for the equivalence, while the last two give two allowable equivalences.

(MAIN_FACULTY DISJOINT BRANCH2_ELEC_ENG_FACULTY)
(BRANCH2_ELEC_ENG_FACULTY CONTAINED IN BRANCH2_FACULTY)
(MAIN_FACULTY DISJOINT BRANCH2_ENG_FACULTY)
(BRANCH1_FACULTY DISJOINT or OVERLAP BRANCH2_ENG_FACULTY)
(BRANCH2_FACULTY DISJOINT or OVERLAP BRANCH2_FACULTY)

4.3 Implementation in PROLOG

Algorithm 4.1 is implemented in PROLOG. In PROLOG, one typically distinguishes between facts and rules. Facts state our static knowledge of the world, whereas a rule allows the deduction of a new fact based on the truth of some other facts. Facts that are considered in the algorithm are the following:

Legend

=: equal
ci: contained in
c: contain
d: disjoint
o: ovelap

Equivalence = (Row Equivalence Column)

Node	1	2	3	4	5
1	=	d	d	ci	ci
2	d	=	o	{d,o}	{d,o}
3	d	o	=	d	d
4	c	{d,o}	d	=	c
5	c	{d,o}	d	ci	=

Figure 4.6 Equivalence Assertion (ER) matrix after applying Algorithm 4.1

(a) Relations in the same cluster are asserted as follows:

```
relations([main_f, branch1_f, branch2_f, branch2_eng_f,
          branch2_ele_eng_f]).
```

The predicate "relations" contains the name of the relation that belongs to the same cluster.

(b) Equivalence between a pair of relations is asserted as follows:

```
equivalence(main_f, branch1_f, o).
equivalence(main_f, branch2_f, d).
equivalence(branch1_f, branch2_ele_eng_f, d).
equivalence(branch2_ele_eng_f, branch2_eng_f, ci).
equivalence(branch2_eng_f, branch2_f, ci).
```

The predicate "equivalence" contains a pair of relations in the first and second arguments, and an equivalence between them in the third argument. For example, equivalence(main_f, branch_f, o) means that the equivalence value between the pair of relations MAIN_FACULTY and BRANCH_FACULTY is overlap.

(C) Before representing the Transitive Rule table in Figure 4.5 in Horn clause, let us consider again the meaning of the table. If for three relations R1, R2 and R3, E1, E2 and E3 are equivalences between R1 and R2, R2 and R3, and R1 and R3, respectively, then the following transitive relationship holds:

(R1 E1 R2) and (R2 E2 R3) --> (R1 E3 R3)

Each of E1, E2 and E3 can be one of the following elements, {=, c, ci, d, o}. The predicate "transitive_rule" represents the equivalence relationship among E1, E2 and E3 as follows:

transitive_rule(E1,E2,E3).

For example, transitive_rule(c,d,[c,d,o]) means that for the three relations R1, R2, and R3, the equivalences between R1 and R2, and R2 and R3, are c (CONTAIN) and d (DISJOINT), respectively, then the equivalence between R1 and R3 should be one of the elements, {c,d,o}. The Transitive Rule table in Figure 4.5 is asserted as follows:

```
transitive_rule(e, e, [e]).
transitive_rule(e, ci, [ci]).
transitive_rule(e, c, [c]).
transitive_rule(e, d, [d]).
transitive_rule(e, d, [o]).

transitive_rule(ci, e, [ci]).
transitive_rule(ci, ci, [ci]).
transitive_rule(ci, c, [e,ci,c,d,o]).
transitive_rule(ci, d, [d]).
transitive_rule(ci, o, [ci,d,o]).

transitive_rule(c, e, [c]).
transitive_rule(c, ci, [e,ci,c,d,o]).
transitive_rule(c, c, [c]).
transitive_rule(c, d, [c,d,o]).
transitive_rule(c, o, [c,o]).
```

```
transitive_rule(d, e, [d]).
transitive_rule(d, ci, [ci,d,o]).
transitive_rule(d, c, [d]).
transitive_rule(d, d, [e,ci,c,d,o]).
transitive_rule(d, o, [ci,d,o]).

transitive_rule(o, e, [o]).
transitive_rule(o, ci, [ci,o]).
transitive_rule(o, c, [c,d,o]).
transitive_rule(o, d, [c,d,o]).
transitive_rule(o, o, [e,ci,c,d,o]).
```

With the "transitive_rule" predicate, we can now represent equivalence by transitive path in the form of a Horn clause as follows:

```
equivalence_by_transitive_path(R1,R2,R3,E3) :-
     equivalence(R1,R2,E1),
     equivalence(R2,R3,E2),
     transitive_rule(E1,E2,E3).
```

Clauses of (a) and (b) constitute the input to the algorithm and are asserted by the system dynamically, whereas clauses of (c) state the static information that is used to derive equivalence by transitive rules and therefore asserted once.

4.3.1 Top Level PROLOG Clauses

The algorithm in Section 4.2 is realized in PROLOG in Figure 4.7. First it reads relations in a cluster and equivalence between a pair of relations. Then it checks the consistency of asserted equivalences. If the asserted equivalences are consistent, then new equivalences are derived from the asserted equivalences; otherwise, the process stops and feeds back the inconsistent equivalence to the designer. The built-in predicate "asserta" is used to accommodate dynamically changed equivalences during derivation of new equivalences. Finally, the built-in predicate "retract" is used to remove from the dynamic databases all clauses whose head matches equivalence.

The predicate "read_data" defined in clause (2) of Figure 4.7 opens the input file, reads data and the data is asserted until end_of_file mark is encountered. The built-in predicate "repeat" always succeeds and can be backtracked an infinite number of times. (p;q) tries p and, if p fails, then tries q. If the data just read does not equal end_of_file, the data is asserted and the built-in predicate "fail" causes a backtrack to "repeat". When data does equal end_of_file, the cut, !, throws away the backtracking history and the input file, F, is closed and returns the input stream back to the user.

The purpose of clauses (3) and (4) of Figure 4.7, is to find the relation pairs whose equivalences are asserted, and

the pairs whose equivalences are not asserted, respectively. In particular, the built-in predicate "findall" is used to collect all the pairs asserted in the equivalence.

```
(1)
top :-
     read_data,
     find_pairs_asserted(AssertedList),
     find_pairs_not_asserted(NotAssertedList),
     check_consistency_of_asserted_equiv(AssertedList),
     derive_new_equiv_from_asserted_equiv(NotAssertedList),
     retract_all(pairs(_)),
     retract_all(equivalence(_,_,_)).

(2)
read_data :-
     set_io(input,F), repeat, read(D),
     (D = end_of_file, !, set_io(input,previous)
     ;
     asserta(D), fail).

(3)
find_pairs_asserted(AssertedList) :-
     findall([R1,R2],equivalence(R1,R2,E),AssertedList).

(4)
find_pairs_not_asserted(NotAssertedList) :-
     relations(RelationList),
     find_pairs(RelationList,NotAssertedList).

find_pairs([],[]).
find_pairs([Relation|Rest],NotAssertedList) :-
     make_pair(Relation,Rest,PairList),
```

```
    find_pairs(Rest,NewNotAssertedList),
    append(PairList,NewNotAssertedList,NotAssertedList).

make_pair(_,[],[]).
make_pair(Relation1,[Relation2|L],PairList) :-
    equivalence(R1,R2,E),
    ((Relation1=R1, Relation2=R2)
    ;
    (Relation1=R2, Relation2=R1)), !,
    make_pair(Relation1,L,PairList).
make_pair(Relation1,[Relation2|L],[PairList|L1]) :-
    PairList = [Relation1,Relation2],
    make_pair(Relation1,L,L1).
```

Figure 4.7 Top Level PROLOG Clauses

4.3.2 Consistency Check of Asserted Equivalences

Once the asserted equivalences have been obtained, the consistency check can be achieved as shown in clause (1) of Figure 4.8. For each pair of relations (Pair) whose equivalence is asserted, all the transitive paths (TransitivePath) are obtained. The asserted equivalence (Eq) is compared with the equivalence generated by transitive path. This process continues recursively until the list of asserted pairs is empty.

Clause (2) of Figure 4.8 is used to obtain all the transitive paths from a pair of relations. First, we get relations (RelationList) in a cluster and subtract a pair of relations (Pair) from relations to obtain the remaining

relations (RestRelation) which are used to generate the transitive path.

In clause (3) of Figure 4.8 we obtain an equivalence from transitive path and compare it with the asserted equivalence. If the asserted equivalence (Eq) is a subset of the equivalence (E3) obtained from the transitive path, then the asserted equivalence is in consistent state; otherwise, the asserted equivalence is inconsistent and an error message is returned. When we obtain equivalence by transitive path as shown in clause (4) of Figure 4.8, the order of a pair of relations should be taken into account. If the order of arguments in a pair of relations needs to be changed, the clause (5) of Figure 4.8 is used. If the equivalence of a pair of relations is one of (e,d,o), then changing the order of arguments does not affect the equivalence. If the equivalences are c and ci, then the equivalences are changed to ci and c, respectively, as a consequence of changing the order of arguments of a pair of relations. For example, for two relations R1 and R2, if R1 contains (c) R2, then it is equivalent to say that R2 is contained in (ci) R1.

```
(1)
check_consistency_of_asserted_equiv([]).
check_consistency_of_asserted_equiv([Pair|L]) :-
     Pair = [R1,R2],
     equivalence(R1,R2,Eq),
     get_transitive_path_from_pair(Pair,TransitivePath),
     check_equiv_from_transitive_path(Eq,TransitivePath),
```

```
     check_consistency_of_asserted_equiv(L).

(2)
get_transitive_path_from_pair(Pair,TransitivePath) :-
     relations(RelationList),
     subtract(RelationList,Pair,RestRelation),
     make_transitive_path(Pair,RestRelation,TransitivePath).

make_transitive_path(Pair,[],[]).
make_transitive_path(Pair,[R2|L1],[TransitivePath|L2]) :-
     Pair = [R1,R3],
     TransitivePath = [R1,R2,R3],
     make_transitive_path(Pair,L1,L2).

(3)
check_equiv_from_transitive_path(Eq,[]).
check_equiv_from_transitive_path(Eq,[TransitivePath|L]) :-
     TransitivePath = [R1,R2,R3], !,
     equivalence_by_transitive_path(R1,R2,R3,E),
     (member(Eq,E)
     ;
     writeln('consistency error')),
     check_equiv_from_transitive_path(Eq,L).
check_equiv_from_transitive_path(Eq,[TransitivePath|L]) :-
     check_equiv_from_transitive_path(Eq,L).

(4)
equivalence_by_transitive_path(Rel1,Rel2,Rel3,Eq) :-
     ((equivalence(Rel1,Rel2,Eq1),
       equivalence(Rel2,Rel3,Eq2))
     ;
     (switch_arg(Rel1,Rel2,Eq1),
      equivalence(Rel2,Rel3,Eq2))
     ;
     (equivalence(Rel1,Rel2,Eq1),
```

```
     switch_arg(Rel2,Rel3,Eq2))
    ;
    (switch_arg(Rel1,Rel2,Eq1),
     switch_arg(Rel2,Rel3,Eq2))),
    transitive_rule(Eq1,Eq2,Eq).

(5)
switch_arg(Rel1,Rel2,e) :- equivalence(Rel2,Rel1,e).
switch_arg(Rel1,Rel2,d) :- equivalence(Rel2,Rel1,d).
switch_arg(Rel1,Rel2,o) :- equivalence(Rel2,Rel1,o).
switch_arg(Rel1,Rel2,ci) :- equivalence(Rel2,Rel1,c).
switch_arg(Rel1,Rel2,c) :- equivalence(Rel2,Rel1,ci).
```

Figure 4.8 PROLOG Clauses for a Consistency Check of Asserted Equivalences

4.3.3 Derivation of New Equivalences from Asserted Equivalences

Once the pairs that are not asserted are obtained, the new equivalence can be derived from the asserted equivalences as defined in clause (1) of Figure 4.9. This derivation continues recursively until no more equivalence is obtained. The termination is done by comparing the unasserted pairs before and after the derivation process. If the unasserted pairs before derivation (PairList) are the same as those after derivation (NewPairList), the process is terminated. Otherwise, it continues recursively with new unasserted pairs.

In clause (2) of Figure 4.9, all the transitive paths (TransitivePath) from an unasserted pair (Pair) are obtained.

When all the equivalences are found for all the possible transitive paths, they are intersected. The result of intersection is returned to the user. If the intersection of equivalences (Eq) is a single value, then the equivalence is asserted. Otherwise, it continues with the remaining unasserted pairs. Also, the pair just asserted (Pair) is subtracted from the current list of unasserted pairs (PairsList). Then, new unasserted pairs (NewPairList) is asserted. This new one is used in the next process. This process continues until the unasserted pair is an empty list.

Clause (3) of Figure 4.9 is used to obtain intersection of equivalences of pairs built by transitive paths. Whenever new equivalence (E) is obtained, it is intersected with the previous result (Eq); and continues recursively with the intersected equivalence (E1). Initially, Eq is set to all the possible equivalences, i.e., (e,ci,c,d,o). If there is no equivalence by transitive path, then the process continues recursively with the equivalence (Eq) unchanged. The recursion terminates when the list of transitive paths is empty and then the equivalences intersected so far are returned.

```
(1)
derive_new_equiv_from_asserted_equiv(PairList):-
     asserta(pairs(PairList)),
     derive_new_equivalence(PairList),
     pairs(NewPairList),
```

```
     ((PairList = NewPairList), !
     ;
     derive_new_equiv_from_asserted_equiv(NewPairList)).

(2)
derive_new_equivalence([]).
derive_new_equivalence([Pair|L]) :-
     get_transitive_path_from_pair(Pair,TransitivePath),
     intersection_of_possible_equiv(TransitivePath,Eq),
     Pair = [R1,R2],
     writeln(equivalence(R1,R2,Eq)),
     (length(Eq,1), !, Eq = [SingleEq],
     asserta(equivalence(R1,R2,SingleEq)),
     pairs(PairsList), delete(Pair,PairsList,NewPairList),
     asserta(pairs(NewPairList)),
     derive_new_equivalence(L)
     ;
     derive_new_equivalence(L)).

(3)
intersection_of_possible_equiv(TransitivePath,Eq):-
     intersection_of_equiv(TransitivePath,[e,ci,c,d,o],Eq).

intersection_of_equiv([],Eq,Eq).
intersection_of_equiv([TransitivePath|L],Eq,NewEq) :-
     TransitivePath = [R1,R2,R3],
     (equivalence_by_transitive_path(R1,R2,R3,E), !,
     intersect(E,Eq,E1),
     intersection_of_equiv(L,E1,NewEq)
     ;
     intersection_of_equiv(L,Eq,NewEq)).
```

Figure 4.9 PROLOG Clauses for Derivation of New Equivalences from Asserted Equivalences

In this chapter we introduced schema integration methodology used in this dissertation and showed five types of equivalences (EQUAL, CONTAIN, CONTAINED_IN, OVERLAP, DISJOINT) among component relations to be used in defining a global view. The identification of these equivalences is important to generate a global schema from component schemas. Along this line, an algorithm has been developed to aid the designer in deriving new equivalences from partially known equivalences as well as to check consistency of equivalences that are previously specified. We showed the implementation of the algorithm in PROLOG.

In the next chapter we consider an extended SQL (ESQL) to provide the user with set operations on component relations related each other by the five types of equivalences.

CHAPTER 5
ESQL: AN EXTENDED SQL FOR THE RELATIONAL MODEL SUPPORTING FEDERATED DATABASES

Most prior work on query language for federated databases has concentrated on querying the information against the global schema. However, it does not consider the relationship between occurrences of an entity type and occurrences of its subentity types (e.g., information about "intersection" occurrences between two overlapped subentity types or "difference" occurrences by subtracting occurrences of a particular subentity type from those of the entity type). Suppose we integrate the databases of the public university in a state and assume some faculties are working in several universities. One might wish to know which faculty is working in several universities. To answer this type of queries, it is necessary to provide set operations on the component relations, which are related to each other in terms of equivalences among them.

Although conventional SQL allows set operations on the relations that are union-compatible, it requires several subqueries and a procedural specification to relate those subqueries. However, the set operations in ESQL shown below are expressed nonprocedurally in a single statement.

In Section 5.1 and 5.2, we propose a data model for schema integration and ESQL for supporting set operations on component relations. In Section 5.3, we consider implementation aspects of set operations. In the last section, we present how domain information such as equivalences among component relations are utilized to facilitate the processing of the set operations.

5.1 The Data Model

As described in Chapter 4, there are five types of equivalences among relations. Once these equivalences are known among the relations, the relations belonging to the same cluster can be integrated to form a generalized relation. In this case, the generalized relation plays a role as a "global view" relation of its "component" relations. We call a generalized relation a global view relation and the relations constituent of a global view relation component relations. A global view relation can be a component relation at a higher level. The attributes that are common to component relations would be the attributes of a global view relation. The names of component relations are also specified in the global view relation. By introducing the concept of a global view relation into the relational model, we are able to perform set operations on the component relations that are present in the global view relation. The global view relation is a

metarelation in a sense that it encompasses the relations that participate in a global view relation. However, it is reduced to a normal relation unless the name of its component relations are explicitly specified.

The names of the global view relation and component relations can be used to denote tuple variables in the FROM clause of SQL. In general, the FROM clause syntax is given as follows. In this notation, [] denotes one or zero occurrences and symbols enclosed in quotes denote literals.

<Global view relation> ['AS' <Component relation> | <FunctionName> '(' <Relation> ',' <Component relation> ')']
<Relation>: <Global view relation> | <Component relation>
<FunctionName>: UNION | DIFF | INTS

The "AS" construct is used for a global view relation to provide various functions of its component relations. Unless stated explicitly, the global view relation implies the union of its component relations.

As an example of the data model for schema integration, consider two local relational schemas in Figure 5.1 and Figure 5.2. They represent information related to a university database, either on the main campus or on a branch campus. Note that we use only one branch campus rather than branch1 and branch2 campuses taken in Chapter 3. For simplicity, type declarations are omitted. For FACULTY relations in both databases, the first one describes main campus faculties, who

have a yearly salary and have an office on the main campus. The second one describes branch faculties, who have a monthly salary, but do not have an office. The same person can be both a main campus faculty and a branch campus faculty. The global schema gives an integrated view of faculties, in which MAIN_FACULTY and BRANCH_FACULTY are seen as component relations of a generalization hierarchy having FACULTY as a global view relation. The FACULTY has three single valued attributes: Name, Salary, and Phone. The information about offices of faculties is assumed to be irrelevant for the global view. The relations MAIN_UNDERGRADUATE and BRANCH_UNDERGRADUATE, and MAIN_GRADUATE and BRANCH_GRADUATE are integrated into the global view relations UNDERGRADUATE and GRADUATE, respectively. The relations UNDERGRADUATE and GRADUATE are integrated into the global view relation STUDENT.

MAIN_FACULTY(Name,Salary,Office#)
MAIN_OFFICE(Office#,Phone)
MAIN_UNDERGRADUATE(SS#,UndergradName,GPA,Class,Major)
MAIN_GRADUATE(SS#,GradName,GPA,DegreeProgram,Major)
MAIN_ADVISOR(Name,SS#)
MAIN_ENROLLMENT(SS#,CourseName,Grade)
MAIN_COURSE(CourseName,Section#,Time)

Figure 5.1 Local Schema at Site 1

BRANCH_FACULTY(Name,Salary,Phone)
BRANCH_UNDERGRADUATE(SS#,UndergradName,GPA,Class)
BRANCH_GRADUATE(SS#,GradName,GPA)

```
BRANCH_ENROLLMENT(SS#,CourseName,Grade)
BRANCH_COURSE(CourseName,Section#,Time)
```

Figure 5.2 Local Schema at Site 2

```
FACULTY(Name,Salary,Phone,[Main_faculty,Branch_faculty])
UNDERGRADUATE(SS#,UndergradName,GPA,Class,
              [Main_undergraduate,Branch_undergraduate])
GRADUATE(SS#,GradName,GPA,[Main_graduate,Branch_graduate])
STUDENT(SS#,StudName,GPA,[Undergraduate,Graduate])
ENROLLMENT(SS#,CourseName,Grade,[Main_enrollment,
           Branch_enrollment])
COURSE(CourseName,Section#,Time)
```

Figure 5.3 Global Schema

The global schema is shown in Figure 5.3. The lists in the bracket specify the names of the component relations that participate in the global view relation. This schema representation is similar to GEM [Zani83]. The ordinary operations such as comparison (=,≠,<,≤,>,≥) are not applicable to the "relation-name" field, while set operations such as union, intersection and difference are allowed on this field.

There are two options for the visibility of integrated schema depending upon whether local schemas are accessible to the user. One option allows the user to access a global schema as well as local schemas. By allowing the user to access local schemas, the user can obtain the field of component schema that cannot be integrated in the global

schema (e.g., Office# in Figure 5.1). However, when we join two relations in both global and component schemas, sophisticated query processing is required. This option is not recommended if all the fields in component schemas are represented in global schema. The other option allows the user to access only the global schema. This option is recommended for novice users who want to avoid the complexity of many similar attributes in global and component schemas. The decision whether local schemas should be visible to the user is ultimately a matter of system policy.

5.2 The Query Language, Extended SQL (ESQL)

ESQL is designed to be a generalization of SQL. Whenever the relations-name field in the global view relation is not used, the global view relation is identical to the normal relation and the syntax of the query is reduced to a conventional SQL. For example, the following query that retrieves the faculty whose salary is over $50,000 uses FACULTY as a normal relation.

```
SELECT    f.name
FROM      faculty f
WHERE     f.salary > 50,000
```

Figure 5.4 Find the faculty whose salary is over $50,000.

The global view relation FACULTY is the union of its component relations. Thus the same query can also be generated using component relations as follows:

```
SELECT    f.name
FROM      f IS faculty AS UNION(main_faculty,branch_faculty)
WHERE     f.salary > 50,000
```

Figure 5.5 Same as Figure 5.4

In relational calculus the query is defined as follows:

$$
\begin{aligned}
\{t.name \mid\ & (t \in main\_faculty \wedge (\nexists u)\ (\ u \in branch\_faculty \\
& \qquad \wedge (u.name = t.name)) \\
& \qquad \wedge (t.salary > 50{,}000)) \\
& \vee (t \in branch\_faculty \wedge (\nexists u)\ (\ u \in main\_faculty \\
& \qquad \wedge (u.name = t.name)) \\
& \qquad \wedge (t.salary * 12 > 50{,}000)) \\
& \vee (\exists u)\ (\exists v)\ (\ u \in main\_faculty \wedge v \in branch\_faculty \\
& \qquad \wedge (u.name = v.name) \\
& \qquad \wedge (t.name = u.name) \\
& \qquad \wedge (u.salary + v.salary * 12 > 50{,}000))\}
\end{aligned}
$$

The equivalent SQL is given as follows:

```
SELECT    m.name
FROM      main_faculty m
WHERE     NOT EXISTS  (SELECT  *
                       FROM    branch_faculty b
                       WHERE   m.name = b.name)
AND       m.salary > 50,000
```

```
UNION
SELECT    b.name
FROM      branch_faculty b
WHERE     NOT EXISTS  (SELECT  *
                       FROM    main_faculty m
                       WHERE   b.name = m.name)
AND       b.salary*12 > 50,000
UNION
SELECT    m.name
FROM      main_faculty m
WHERE     EXISTS  (SELECT  *
                   FROM    branch_faculty b
                   WHERE   m.name = b.name
                   AND     m.salary + b.salary*12 > 50,000)
```

As we see in the above example, the query in SQL requires several subqueries and the user has to know the details of the integration involved. However, in ESQL, the query is a single statement. When a query is invoked, the integration represented internally is mapped to operations of component relations.

If we wish to retrieve the salary of the main_faculty whose name is "Smith" in the global view, we can write the query as follows:

```
SELECT    f.salary
FROM      f IS faculty AS main_faculty
WHERE     f.name="Smith"
```

Figure 5.6 Find the salary of the main_faculty whose name is "Smith" in the global view.

or simply put in the following way without an explicit declaration of tuple variable:

```
SELECT    salary
FROM      faculty AS main_faculty
WHERE     name="Smith"
```

Figure 5.7 Same as Figure 5.6

The query "List the names of faculty that is both at the main and branch campuses and earns more than $50,000 a year" would be:

```
SELECT    f.name
FROM      f IS faculty AS INTS(main_faculty,branch_faculty)
WHERE     f.salary > 50,000
```

Figure 5.8 Find the faculty who is both at the main and branch campuses and earn more than $50,000 a year.

In relational calculus the query is defined as follows:

$$\{t.name \mid (\exists u)\ (\exists v)\ (\ u \in main\_faculty \wedge v \in branch\_faculty \wedge (u.name = v.name) \wedge (t.name = u.name) \wedge (u.salary + v.salary * 12 > 50{,}000))\}$$

For another example, to query the faculty who works only at the main_campus one can write:

```
SELECT    f.name
FROM      f IS faculty AS DIFF(main_faculty,branch_faculty)
```

Figure 5.9 Find the faculty who works only at the main campus.

We can define the query in relational calculus as follows:

$$\{t.name \mid (t \in main\_faculty \wedge (\nexists u)\ (u \in branch\_faculty \wedge (u.name = t.name)) \wedge (t.salary > 50{,}000))\}$$

If we want to know what campus faculty "Smith" belongs to, the query can be written as follows:

```
SELECT    faculty*
FROM      faculty f
WHERE     f.name = "Smith"
```

Figure 5.10 Find the campus in which faculty "Smith" works.

The * operator specifies how the global view relation is composed of from its component relations in the global schema. By allowing the component relations in the global schema, we can access the field of component schema that cannot be integrated in the global schema (e.g., office number in the

above example). However, even though both the attribute salary in MAIN_FACULTY (Figure 5.1) and FACULTY (Figure 5.3) is accessible to the user, the semantics is different. For example, if we want to retrieve salary of main_faculty whose name is "Smith" in the local schema, it will be:

```
SELECT    salary
FROM      main_faculty
WHERE     name = "Smith"
```

Figure 5.11 Find the salary of the main_faculty whose name is "Smith" in the local view.

If "Smith" happens to work both at the main and at the branch campuses, Figure 5.11 returns the salary received only at the main campus, while Figure 5.6 returns the salary computed at both campuses.

Set Comparison Operator

ESQL also provides set comparison operators to process the component relations belonging to the same cluster. These operators are not applicable to relation tuples, but to component relations. The results of these operations are obtained from the Equivalence Assertion (EA) table given in Chapter 4.

=	equals
!=	not equal
⊇	contains
⊆	is contained in
!!	is disjoint
++	overlaps

Aggregate Functions and Grouping

As we mentioned before, the integrated schema is reduced to conventional relational model if we do not use the relation-name field specifying the component relations participating in the global view relation. However, using the relation-name field in the global view relation, we can easily invoke a query about the component relations in a single statement. In conventional SQL, the "GROUP-BY" clause is used to group the tuples that have the same value of some attributes. Beside the facility provided by conventional SQL, the "GROUP-BY" clause in ESQL is also used to group the component relations in the global view relation and functions such as AVG, SUM, COUNT, MAX, and MIN are applied to each component relation independently. For example, if we want to know the average of GPA for undergraduate students in each campus, the query will be as follows:

```
SELECT    <COMPONENT>, AVG(gpa)
FROM      undergraduate
GROUP BY  <COMPONENT>
```

Figure 5.13 Find the average GPA for undergraduate students in each campus

If the global view relation UNDERGRADUATE is composed of the component relations MAIN-UNDERGRADUATE and BRANCH-UNDERGRADUATE, the answer to the query will be in the form as follows:

COMPONENT(undergraduate)	AVG(gpa)
Main_undergraduate	...
Branch_undergraduate	...

5.3 Algorithms for Set Operations

When we generate a global view definition from component relations, the possible cases to consider in the global view definition are obtained from the identification of equivalences among the component relations. For example, if two component relations MAIN_FACULTY and BRANCH_FACULTY overlap, then we have to consider three cases in defining the global view relation FACULTY: MAIN_FACULTY DIFFERENCE BRANCH_FACULTY, BRANCH_FACULTY DIFFERENCE MAIN_FACULTY, and

MAIN_FACULTY INTERSECTION BRANCH_FACULTY. The set operations (i.e., UNION, DIFFERENCE, and INTERSECTION) on component relations are then obtained from the cases defined in the global view. For example, if we want to perform a union operation on the component relations MAIN_FACULTY and BRANCH_FACULTY, the required cases consist of all the three cases, namely, MAIN_FACULTY DIFFERENCE BRANCH_FACULTY, BRANCH_FACULTY DIFFERENCE MAIN_FACULTY, and MAIN_FACULTY INTERSECTION BRANCH_FACULTY.

To perform the set operations in general, we take an extreme case which subsumes all other cases. That is, all of the component relations overlap. If there are r component relations and all of them overlap, the minimally required cases for a global view definition will be 2^r-1. For example, if three relations overlap, then we have to consider $2^2-1=3$ cases to define the global view.

Given r component relations A_1, A_2,..., A_r that overlap each other, we define A_i^0, $(A_i \cap A_{i+1})^0$,..., $(A_1 \cap A_2 \cap \ldots \cap A_r)^0$ in the following way. A_i^0 is composed of the tuples that belong to the relation A_i only.

$$A_i^0 = \{x \mid x \in A_i \wedge x \notin A_1 \wedge x \notin A_2 \wedge \ldots \wedge x \notin A_{i-1} \wedge x \notin A_{i+1} \wedge \ldots x \wedge A_r\}$$

Likewise, $(A_i \cap A_{i+1})^0$ is composed of the tuples that are intersection of the two component relations A_i and A_{i+1} only.

$(A_i \cap A_{i+1})^0 = \{x \mid x \in (A_i \cap A_{i+1}) \wedge x \notin A_1 \wedge x \notin A_2 \wedge \ldots \wedge x \notin A_{i-1} \wedge x \notin A_{i+2} \wedge \ldots \wedge x \notin A_r\}$

...

$(A_1 \cap A_2 \cap \ldots \cap A_r)^0 = \{x \mid x \in (A_1 \cap A_2 \cap \ldots \cap A_r)\} \equiv A_1 \cap A_2 \cap \ldots \cap A_r$

Then $S_1, S_2, \ldots, S_n$ defined below should be the minimum view definitions to be generated manually considering the semantics of the attributes of the component relations.

$S_1 = \{A_i^0 \mid 1 \le i \le r\}$

$S_2 = \{(A_1 \cap A_i)^0 \mid 2 \le i \le r, (A_2 \cap A_i)^0 \mid 3 \le i \le r, \ldots, (A_{r-1} \cap A_r)^0\}$

...

$S_n = \{A_1 \cap A_2 \cap \ldots \cap A_r\}$

Let us define S as the whole set: $S \triangleq S_1 \cup S_2 \cup \ldots \cup S_n$. Then, each component relation A_1, A_2, ..., A_r and set operations on them are computed as functions of the elements of set S in the following way. The first function generates A_i^0, $(A_i \cap A_{i+1})^0, \ldots, (A_1 \cap A_2 \cap \ldots \cap A_r)^0$ corresponding to a component relation Ai.

```
FUNCTION ElementsOfViewDefinition(S:set, Ai:char): set;
   BEGIN
      T := {};
      FOR each s∈S
         DO IF s contains Ai
               THEN adds s to T;
      ElementsOfViewDefinition := T;
   END;
```

```
FUNCTION Union(Vi:char, Vj:char): set;

   BEGIN
      Ai := ElementsOfViewDefinition(S,Vi);
      Aj := ElementsOfViewDefinition(S,Vj);
      T := Ai;
      FOR each a∈Ai
         DO IF a∉Aj
               THEN T := T ∪ a;
      Union := T;
   END;


FUNCTION Intersection(Vi:char, Vj:char): set;

   BEGIN
      Ai := ElementsOfViewDefinition(S,Vi);
      Aj := ElementsOfViewDefinition(S,Vj);
      T := ();
      FOR each a∈Ai
         DO IF a∈Aj
               THEN T := T ∪ a;
      Intersection := T;
   END;


FUNCTION Difference(Vi:char, Vj:char): set;

   BEGIN
      Ai := ElementsOfViewDefinition(S,Vi);
      Aj := ElementsOfViewDefinition(S,Vj);
      T := ();
      FOR each a∈Ai
         DO IF a∉Aj
               THEN T := T ∪ a;
      Difference := T;
   END;
```

To illustrate the algorithms, consider three component relations A_1, A_2 and A_3 that happens to overlap each other as shown in Figure 5.12. If the user wants to know which tuples belong only to the relations A_1, then the set operations are specified by DIFF(A_1,UNION(A_2,A_3)). All of the view

definitions required for the three relations are given as follows:

$S = \{A_1^0, A_2^0, A_3^0, (A_1 \cap A_2)^0, (A_1 \cap A_3)^0, (A_2 \cap A_3)^0, (A_1 \cap A_2 \cap A_3)\}$

Now A_1, A_2 and A_3 are obtained from the above algorithm as follows:

$A_1 = \{A_1^0, (A_1 \cap A_2)^0, (A_1 \cap A_3)^0, (A_1 \cap A_2 \cap A_3)\}$

$A_2 = \{A_2^0, (A_1 \cap A_2)^0, (A_2 \cap A_3)^0, (A_1 \cap A_2 \cap A_3)\}$

$A_3 = \{A_3^0, (A_1 \cap A_3)^0, (A_2 \cap A_3)^0, (A_1 \cap A_2 \cap A_3)\}$

Then, $\text{UNION}(A_2, A_3) = \{A_2^0, (A_1 \cap A_2)^0, (A_1 \cap A_3)^0, (A_2 \cap A_3)^0, (A_1 \cap A_2 \cap A_3)\}$. Thus, $\text{DIFF}(A_1, \text{UNION}(A_2, A_3)) = \{A_1^0\}$, which corresponds to the portion of the tuples belonging only to the relation A_1 as shown in Figure 5.12.

5.4 Generalization (Integration) Integrity Constraints

Thus far, we allow all combinations of set operations on component relations participating in the generalization. However, if we consider the application domain (e.g., how the component relations participate in the global view relation), as it was done in Semantic Query Optimization [Chak90], performance improvements associated with set operations can be achieved. For example, suppose three relations, MAIN_FACULTY, BRANCH1_FACULTY AND BRANCH2_FACULTY, are integrated into a global view relation FACULTY and their relationship is illustrated in Figure 5.13. Suppose one wishes to know

whether there are faculties that work at both branch1 and branch2. Then the result will be null after processing set operations and accessing local databases. But if we explicitly specify the generalization integrity constraints associated with this domain, we can obtain an answer immediately without accessing local databases. This relationship is represented in the truth table given in Figure 5.14. The "true" input in the three component relations means that "if" the corresponding component relation is requested and the "true" output in the global view relation faculty means that there exist instances for such request. For example, the 7th row shows that there exist faculties who work both at the main and branch1 campuses. On the other hand, the "false" output in the global view relation means that there are no instances for the corresponding request on the component relations. The 4th row, for example, shows that faculty who works both at the branch1 and branch2 campuses does not exist. Therefore, such requests will result in no instances. The combination of inputs which have a "true" response can represent in the propositional logic in Figure 5.15. This generalization integrity constraint shows how the set of component relation tuples participate in the global view relation.

$$
\begin{aligned}
IC = \; & (\neg main\_f \wedge \neg branch1\_f \wedge branch2\_f) \\
& \vee (\neg main\_f \wedge branch1\_f \wedge \neg branch2\_f) \\
& \vee (main\_f \wedge \neg branch1\_f \wedge \neg branch2\_f) \\
& \vee (main\_f \wedge branch1\_f \wedge \neg branch2\_f)
\end{aligned}
$$

Figure 5.15 Generalization Integrity Constraint in Propositional Logic.

In this chapter we introduced an extended relation model to deal with schema integration and ESQL for supporting set operations on component relations. We developed the algorithms to provide any combination of set operations from minimally generated view definition. We utilized the generalization integrity constraints to obtain an answer immediately for some queries without accessing local databases.

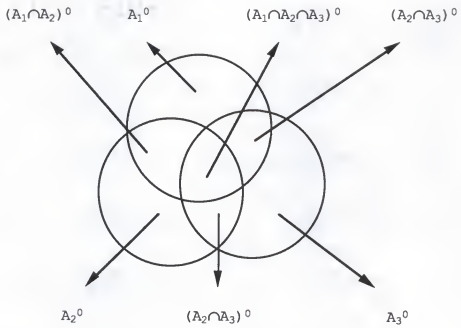

Figure 5.12 Venn Diagram for three Relations

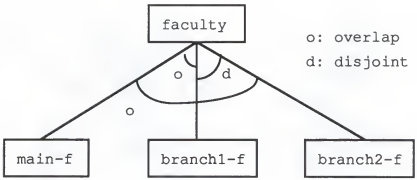

Figure 5.13 Relationship between representative relation, faculty, and its component relations, main_f, branch1_f and branch2_f

component relations			global view relation
main-f	branch1-f	branch2-f	faculty
F	F	F	F
F	F	T	T
F	T	F	T
F	T	T	F
T	F	F	T
T	F	T	T
T	T	F	T
T	T	T	F

Figure 5.14 Generalization Integrity Constraints in Propositional Logic.

CHAPTER 6
GLOBAL VIEW DEFINITION IN HORN CLAUSE

In Chapter 5, we described an extended relational model to support schema integration and an extended SQL for specifying user query against the global schema. In this chapter we show how to define a global schema composed of component schemas in terms of Horn clauses. Also, we show that integrity constraints associated with global schema can be expressed in logic. Therefore, logic is used to completely specify a global schema in terms of component schemas, together with expressing integrity constraints. These integrity constraints are used to optimize a query, which is described in Chapter 7. Section 6.1 introduces the basic terminology related to logic and shows how the global schema is defined. We assume that the databases for integration are structured according to the hierarchical, network and relational data models. That would cover the majority of available databases on mini and mainframe systems. In Sections 6.2-6.3 we illustrate how to convert the network and hierarchical database schemas to the relational views and how to represent them in Horn clauses.

6.1 Global View Definition

In this section we begin with the basic terminology related to logic. To be independent of any particular PROLOG syntax, the notations adopted here are similar to Datalog [Ullm88], which was coined to suggest a version of PROLOG suitable for database systems.

A predicate whose relation is stored in the database is called an extensional database (EDB) relation, while one defined by logical rules is called an intensional database (IDB) relation. We assume that each predicate symbol either denotes an EDB relation or an IDB relation, but not both.

Datalog programs are built from atomic formulas, which are predicate symbols with a list of arguments, e.g., $p(A_1,\ldots,A_n)$, where p is the predicate symbol and $A_1,\ldots,A_n$ are the arguments. An argument in datalog can be either a variable or a constant. We use names beginning with lower case letters for constants and predicate names, while using names beginning with upper case letters for variables. Also, we use the corresponding upper case letter to represent the corresponding relation of a predicate. For example, P represents the relation for the predicate p.

A literal is either an atomic formula or a negated atomic formula. A clause is a disjunction of literals. A Horn clause is a clause with at most one positive literal. A collection of Horn clauses is termed a logic program. A Horn

clause is either (1) a single positive literal, e.g., p(X,Y), which is called a fact, or (2) a positive literal and one or more negative literals, which is called rule. We use PROLOG style for expressing Horn clauses, like

q :- p1, p2, ... ,pn.

for the Horn clause $q \vee \neg p1 \vee \neg p2 \vee \ldots \vee \neg pn$.

In order to show how global schema is derived from local schemas, let us use the schemas in Figure 5.1-5.3. The global schema FACULTY is derived from the MAIN_FACULTY in local schema 1 and the BRANCH_FACULTY in local schema 2. While the attribute Name is trivially derived, problems arise for the attributes Phone and Salary. In particular, for those faculties who are both main_faculty and branch_faculty, it is assumed that: 1) The phone numbers of the main and branch campuses are enumerated in the list form, 2) The annual salary is computed by adding the salary as main_faculty to the salary as branch_faculty computed from the monthly_based salary. For simplicity, let us call the salary of main_faculty Salary1 and the salary of branch_faculty Salary2, respectively. The minimally required Horn clauses for the definition of global view relation, FACULTY, will be three cases, i.e., for tuples belonging to the intersection between MAIN_FACULTY and BRANCH_FACULTY, and to the two differences between them. The derivation rules for faculty is as follows:

```
faculty(Name,Salary,Phone,ints(main,branch)) :-
     main_faculty(Name,Salary1,Office#),
     office(Office#,Main_ph),
     branch_faculty(Name,Salary2,Branch_ph),
     Phone = [Main_ph,Branch_ph],
     Salary = Salary1 + 12*Salary2.

faculty(Name,Salary,Phone,diff(main,branch)) :-
     main_faculty(Name,Salary,Office#),
     office(Office#,Phone),
     branch_faculty(N,_,_),
     Name ≠ N.

faculty(Name,Salary,Phone,diff(branch,main)) :-
     branch_faculty(Name,Salary1,Phone),
     Salary = 12*Salary1,
     main_faculty(N,_,_),
     Name ≠ N.
```

The fourth argument of faculty is used to distinguish the tuples belonging to the three cases, i.e., intersection between MAIN_FACULTY and BRANCH_FACULTY, difference MAIN_FACULTY from BRANCH_FACULTY, and difference BRANCH_FACULTY from MAIN_FACULTY. The derivation rules for the global schema solve all the conflicts presented by the two local schemas. When a query is posed over the global schema, the compilation transforms the query into the corresponding query over the local schemas by using these derivation rules.

In order to facilitate the compilation and analysis of logic programs, rules represented in different forms should be

rectified [Ullm88]. The rules for predicate p are rectified if all their heads are identical, and of the form $p(X_1,\ldots,X_k)$ for distinct variables $X_1,\ldots,X_k$. The above rule set for FACULTY can be rectified as shown in Figure 6.1. In addition to the global schema definition, integrity constraints associated with the global schema and its components schemas are expressed using first order logic in Figure 6.2. The sentence for the first functional dependency in Figure 6.2.a can be read as saying that for any two instances of FACULTY, if the Name values are the same, then the Salary, Phone and Component values must be the same also. In other words, Name uniquely determines Salary, Phone and Component. Similarly, the second and third sentences describe the functional dependencies of component schemas, MAIN_FACULTY and BRANCH_FACULTY, respectively. The inclusion dependency of the sentence of Figure 6.2.b is to say that for each instance of MAIN_FACULTY there exists an instance of OFFICE. In other words, the set of names obtained by projecting MAIN_FACULTY on Office# must be a subset of the set obtained by projecting OFFICE on Office#. The data integration constraints in Figure 6.2.c show which attributes of component schemas are used to generate the attribute of global schema. For example, the effect of the second sentence of Figure 6.2.c is to say that the attribute Salary of FACULTY is obtained from that of both MAIN_FACULTY and BRANCH_FACULTY. In other words, the attribute Salary of global view relation FACULTY depends on

the values of the attribute Salary in both component relations MAIN_FACULTY and BRANCH_FACULTY. In Chapter 7 we will deal with the application of these integrity constraints when simplifying a query.

```
faculty(Name,Salary,Phone,Component) :-
     main_faculty(Name,Salary1,Office#),
     office(Office#,Main_ph),
     branch_faculty(Name,Salary2,Branch_ph),
     Phone = [Main_ph,Branch_ph],
     Salary = Salary1 + 12*Salary2,
     Component = ints(main,branch).

faculty(Name,Salary,Phone,Component) :-
     main_faculty(Name,Salary,Office#),
     office(Office#,Phone),
     branch_faculty(N,_,_),
     Name ≠ N,
     Component = diff(main,branch).

faculty(Name,Salary,Phone,Component) :-
     branch_faculty(Name,Salary1,Phone),
     Salary = 12*Salary1,
     main_faculty(N,_,_),
     Name ≠ N,
     Component = diff(branch,main).
```

Figure 6.1 Global Schema Definition in Horn Clause

(a) Functional Dependencies

```
∀ Name, ∀ Salary1, ∀ Phone1, ∀ Component1,
∀ Salary2, ∀ Phone2, ∀ Component2
(faculty(Name,Salary1,Phone1,Component1),
 faculty(Name,Salary2,Phone2,Component2)
 --> (Salary1=Salary2, Phone1=Phone2, Component1=Component2))

∀ Name, ∀ Salary1, ∀ Phone1, ∀ Salary2, ∀ Phone2
(main_faculty(Name,Salary1,Office#1),
 main_faculty(Name,Salary2,Office#2)
 --> (Salary1=Salary2, Office#1=Office#2))

∀ Name, ∀ Salary1, ∀ Phone1, ∀ Salary2, ∀ Phone2
(branch_faculty(Name,Salary1,Phone1),
 branch_faculty(Name,Salary2,Phone2)
 --> (Salary1=Salary2, Phone1=Phone2))
```

(b) Inclusion Dependency

```
∀ Name, ∀ Salary, ∀ Office#, ∃ Phone
(main_faculty(Name,Salary,Office#)
 --> office(Office#,Phone))
```

(c) Data Integration Dependencies

```
∀ Name1, ∀ Name2, ∃ Name, ∀ Salary1, ∀ Salary2, ∀ Phone1,
∀ Phone2
(main_faculty(Name1,Salary1,Phone1),
 branch_faculty(Name2,Salary2,Phone2)
 --> faculty(Name,Salary,Phone))

∀ Salary1, ∀ Salary2, ∃ Salary, ∀ Name1, ∀ Name2, ∀ Phone1,
∀ Phone2
(main_faculty(Name1,Salary1,Phone1),
```

```
 branch_faculty(Name2,Salary2,Phone2)
 --> faculty(Name,Salary,Phone))

∀ Phone1, ∀ Phone2, ∃ Phone, ∀ Salary1, ∀ Salary2, ∀ Salary,
∀ Name1, ∀ Name2
(main_faculty(Name1,Salary1,Phone1),
 branch_faculty(Name2,Salary2,Phone2)
 --> faculty(Name,Salary,Phone))
```

Figure 6.2 Integrity Constraints in Logic for the Global Schema in Figure 6.1

6.2 Relational View of Network Database Schema

A network database schema consists of two basic components: record types and set types. Record type describes the structure of a number of records that have the same type of information. Each record type is given a name and composed of one or more data items. A set type represents a 1:N relationship between two record types. The record type on the 1-side is called the owner record type and the one on the N-side is called the member record type.

In general, the mapping from a network to a relational schema is based on a one to one correspondence between record types and relations, and between data items and attributes. The major difference between the network and the relational data model is the way of modeling relationships among entities. The relational model represents the relationship between two entities using a relation, whereas the network

model represents the relationship using an access path provided by set types.

Our basic method for translating a network data definition to an equivalent relational data definition is the use of the key in the owner record as a foreign key in its member records [Nava80, Zani79]. Then the navigation through access paths in network databases can be accomplished by equi-joining relations in an equivalent relational data definition. For example, the data structure diagram of the DEPARTMENT record and the EMPLOYEE record and their one-to-many relationship is shown in Figure 6.3.a. The equivalent relational data definition is as follows:

DEPARTMENT(Dname,Location,Manager).
EMPLOYEE(Dname,Eno,Ename,Age,Salary).

The record key department name (Dname) of the owner record DEPARTMENT has been propagated through the set into the member record EMPLOYEE. Department name becomes a foreign key in the EMPLOYEE. An example of the many-to-many relationship can be found in the relationship between two record types, EMPLOYEE and PROJECT. The many-to-many relationship is modeled by identifying two one-to-many relationships using intersection data. The data structure diagram representing the many-to-many relationship between the EMPLOYEE record and the PROJECT record is shown in part (b) of Figure 6.3. The equivalent relational data definition is as follows:

EMPLOYEE(Eno,Ename,Age,Salary).
PROJECT(Pno,Pname,Budget).
WORKS_ON(Eno,Pno,Hours).

Figure 6.4 shows data definition for record and set types of network schema of Figure 6.3.a, which is followed from the syntax in Elmasri and Navathe's textbook [Elma89]. Figure 6.5 shown below illustrates Horn clause representation of the relational view definition constructed from the network schema in Figure 6.3.a. Figure 6.6 shows the integrity constraints associated with relational view. In Figure 6.5 the relation DEPARTMENT is defined as the owner record type DEPARTMENT_RECORD. But the relation EMPLOYEE is defined with the owner record type DEPARTMENT_RECORD, the member record type EMPLOYEE_RECORD and the set type DEPT_EMP_SET that links the instances of DEPARTMENT and EMPLOYEE record types.

```
department(Dname,Location,Manager) :-
     department_record(Dname,Location,Manager).

employee(Dname,Eno,Ename,Age,Salary) :-
     department_record(Dname,_,_)@Tid1,
     employee_record(Eno,Ename,Age,Salary)@Tid2,
     dep_emp_set(Tid1,Tid2)
```

Figure 6.5 Relational View Definition in Horn Clause

In network model, record instances can be distinguished by a system-generated tuple identifier (tid) which uniquely

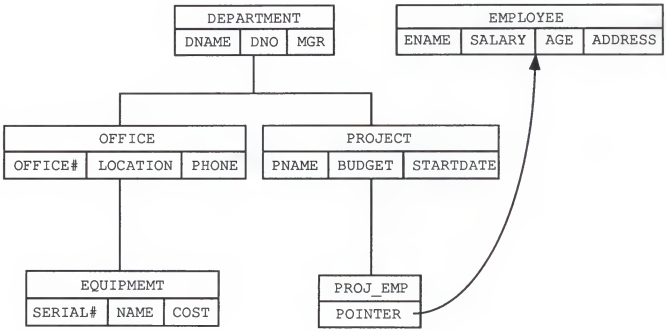

Figure 6.13.(a) A Hierarchical Schema Diagram

```
HIERARCHIES = HIERARCHY1, HIERARCH2

RECORD
  NAME = DEPARTMENT
  TYPE = ROOT OF HIERARCH1
  DATA ITEMS =
    DNAME  CHARACTER 15
    DNO    INTEGER
    MGR    CHARACTER 15
  KEY = DNAME

RECORD
  NAME = EMPLOYEE
  TYPE = ROOT OF HIERARCH2
  DATA ITEMS =
    ENAME       CHARACTER 15
    SALARY      CHARACTER 10
    AGE         INTEGER
    ADDRESS     CHARACTER 30
  KEY = ENAME
```

Figure 6.13.(b) Declarations for the hierarchical schema in Figure 6.13.(a) (Syntax as per Elmasri and Navathe's Book [Elma89]) (Continued)

identifies the record instance. If a record type has a key, then its tid is equal to the key of that record. To refer to a tuple identifier of a record instance, we use the infix predicate "@" which takes a predicate as its left argument and a corresponding tid as the right one and succeeds when the right argument refers to the left one. The set type is then represented in relational view as a join predicate that connects instances of owner and member types. In other words, for each owner-member pair of record instances linked by set type, set type can be considered as a predicate over its arguments consisting of the owner and member tids.

(a) Functional Dependencies

```
∀ Dname, ∀ Location1, ∀ Manager1, ∀ Location2, ∀ Manager2
(department(Dname,Location1,Manager1),
 department(Dname,Location2,Manager2)
 --> (Location1=Location2, Phone1=Phone2))

∀ Dname1, ∀ Eno, ∀ Ename1, ∀ Age1, ∀ Salary1,
∀ Dname2, ∀ Ename2, ∀ Age2, ∀ Salary2
(employee(Dname1,Eno,Ename1,Age1,Salary1)
 employee(Dname2,Eno,Ename2,Age2,Salary2)
 --> (Dname1=Dname2, Ename1=Ename2,
      Age1=Age2, Salary1=Salary2))

∀ Dname, ∀ Location1, ∀ Manager1, ∀ Location2, ∀ Manager2
(department_record(Dname,Location1,Manager1),
 department_record(Dname,Location2,Manager2)
 --> (Location1=Location2, Phone1=Phone2))
```

```
∀ Eno, ∀ Ename1, ∀ Age1, ∀ Salary1,
∀ Ename2, ∀ Age2, ∀ Salary2
(employee_record(Eno,Ename1,Age1,Salary1),
 employee_record(Eno,Ename2,Age2,Salary2)
 --> (Ename1=Ename2,Age1=Age2,Salary1=Salary2))
```

(b) Inclusion Dependency

```
∀ Dname, ∀ Eno, ∀ Ename, ∀ Age, ∀ Salary,
∃ Location, ∃ Manager
(employee(Dname,Eno,Ename,Age,Salary)
 --> department(Dname,Location,Manager)
```

Figure 6.6 Integrity Constraints in Logic for Schema in Figure 6.5

The query, shown in Figure 6.7, issued against the relational view is to retrieve the following information: Find the department name and its manager's name with whom employee John is working. This query is converted to the canonical logic query in Figure 6.8. The query processor transforms the query into a network schema shown in Figure 6.9. If we simplify the query using the integrity constraints shown in Figure 6.6.2, we could obtain the query in Figure 6.10. Now this query is composed of record and set types of the network schema together with selection and projection on the record types. From this information, we can translate into the network query language.

```
SELECT    dname, manager
FROM      department, employee
WHERE     department.dname=employee.dname
AND       employee.ename=john
```

Figure 6.7 Find the department name and its manager's name with whom employee John is working.

```
query(Dname,Manager) :-
     department(Dname,_,Manager),
     employee(Dname,_,john,_,_).
```

Figure 6.8 Equivalent logic query of Figure 6.4

```
query(Dname,Manager) :-
     department_record(Dname,_,Manager),
     department_record(Dname,_,_)@Tid1,
     employee_record(_,john,_,_)@Tid2,
     dep_emp_set(Tid1,Tid2).
```

Figure 6.9 Query translation into network schema

```
query(Dname,Manager) :-
     department_record(Dname,_,Manager)@Tid1,
     employee_record(_,john,_,_)@Tid2,
     dep_emp_set(Tid1,Tid2).
```

Figure 6.10 Simplified query of Figure 6.9

6.3 Relational View of Hierarchical Database Schema

A hierarchical database schema is based on a tree data structure. It consists of a number of record types and parent-child relationship types. A record type is given a name and composed of one or more fields (or data items). A parent-child relationship type (PCR type) is a 1:N relationship between two record types. The record type on the 1-side is called the parent record type and the one on the N-side is called the child record type of the PCR type. Note that PCR types do not have a name in the hierarchical model. However, a PCR type connecting a parent record to a child record has a certain meaning to a database designer. Consider a hierarchical schema shown in Figure 6.11. There are three record types, DEPARTMENT, OFFICE and PROJECT, and two PCR types, (DEPARTMENT-OFFICE) and (DEPARTMENT-PROJECT). Each occurrence of (DEPARTMENT-PROJECT) PCR type relates one department record to the records of the many (zero or more) projects that belong to that department.

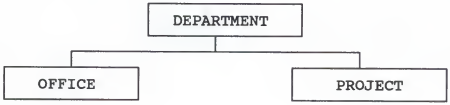

Figure 6.11 A Hierarchical Schema

When the data structure diagram is a tree, data can be represented in the hierarchical database schema in a straightforward way. However, problems arise when representing many-to-many relationships. Consider an M:N relationship between two record types, PROJECT and EMPLOYEE, shown in Figure 6.12.a, where we placed PROJECT as the parent node and EMPLOYEE as the child node. With this representation, we encounter two problems. First, there may be much duplication of data. Since (PROJECT-EMPLOYEE) PCR type is a many-to-many relationship, each EMPLOYEE record must be repeated for each PROJECT record where that employee works for. Record duplication can lead to inconsistencies when maintaining duplicate copies of the same record. Second, it is a complex process to find all the projects for which an employee works, because the whole database may have to be searched.

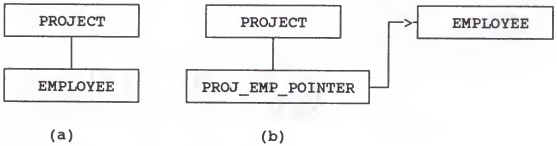

Figure 6.12 M:N relationship without virtual record type (a) and with virtual record type (b)

To get around the problem, the concept of a virtual record type [Elma89] is used. The idea is that more than one hierarchical schema is included in the hierarchical database schema and pointers are used to link two hierarchical schemas. Figure 6.12.b shows the M:N relationship between EMPLOYEE and PROJECT using virtual records. Because each employee record appears only once as a parent record in its tree, the duplication problem is now solved. A pointer in each PROJECT record to its EMPLOYEE record helps find the employees who works for a project, but pointers in virtual PROJ_EMP_POINTER records to PROJECT parent do not help find the projects for which a given employee works. A more general way to represent M:N relationships in the hierarchical model, say between record types A and B, is to make virtual record B a child of record A and virtual record A a child of record B.

Mappings between hierarchical and relational schemas are similar to those between network and relational schemas. Mapping from a hierarchical to a relational model is based on a one to one correspondence between record types and relations, and between fields and attributes. In case of root record type, the key in the relational schema is equivalent to the key in the hierarchical schema. Except for the root record type, the key in the relational schema is composed of the key in the corresponding hierarchical schema together with the key propagated from its ancestors' records. The key of virtual record type consists of the keys in two hierarchical

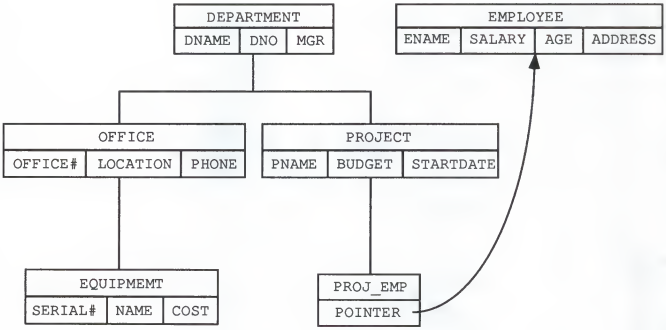

Figure 6.13.(a) A Hierarchical Schema Diagram

```
HIERARCHIES = HIERARCHY1, HIERARCH2

RECORD
  NAME = DEPARTMENT
  TYPE = ROOT OF HIERARCH1
  DATA ITEMS =
    DNAME  CHARACTER 15
    DNO    INTEGER
    MGR    CHARACTER 15
  KEY = DNAME

RECORD
  NAME = EMPLOYEE
  TYPE = ROOT OF HIERARCH2
  DATA ITEMS =
    ENAME       CHARACTER 15
    SALARY      CHARACTER 10
    AGE         INTEGER
    ADDRESS     CHARACTER 30
  KEY = ENAME
```

Figure 6.13.(b) Declarations for the hierarchical schema in Figure 6.13.(a) (Syntax as per Elmasri and Navathe's Book [Elma89]) (Continued)

```
RECORD
  NAME = OFFICE
  PARENT = DEPARTMENT
  CHILD NUMBER = 1
  DATA ITEMS =
    OFFICE#      INTEGER
    LOCATION     CHARACTER 15
    PHONE        CHARACTER 10
  KEY = OFFICE#


RECORD
  NAME = PROJECT
  PARENT = DEPARTMENT
  CHILD NUMBER = 2
  DATA ITEMS =
    PNAME        CHARACTER 15
    BUDGET       INTEGER
    STARTDATE    CHARACTER 9
  KEY = PNAME


RECORD
  NAME = EQUIPMENT
  PARENT = OFFICE
  CHILD NUMBER = 1
  DATA ITEMS =
    SERIAL#      INTEGER
    NAME         CHARACTER 15
    COST         INTEGER
  KEY = SERIAL#


RECORD
  NAME = PROJ_EMP
  PARENT = PROJECT
  CHILD NUMBER = 1
  DATA ITEMS =
    POINTER      POINTER WITH VIRTUAL PARENT = EMPLOYEE
```

Figure 6.13.(b) (Continued) Declarations for the hieracical schema in Figure 6.13.(a)

schemas. Navigation through access paths in hierarchical databases can be done by the join operation in the relational schema using foreign keys propagated from ancestors' records. Figure 6.13 shows the data definition for record and virtual record types of the hierarchical schema based on the syntax of [Elma89]. The equivalent relational schema of the hierarchical schema in Figure 6.13 is given in Figure 6.14. Figure 6.15 and Figure 6.16 show the derivation rules for the equivalent relational schema and its integrity constraints, respectively.

```
DEPARTMENT(Dname,Dno,Manager)
OFFICE(Dname,Office#,Location,Phone)
EQUIPMENT(Dname,Office#,Serial#,Name,Cost)
PROJECT(Dname,Pname,Budget,Startdate)
EMPLOYEE(Ename,Salary,Address)
PROJECT_EMPLOYEE(Dname,Pname,Ename).
```

Figure 6.14 Relation Schema constructed from Hierarchical Schema in Figure 6.13

```
department(Dname,Dno,Manager) :-
     department_record(Dname,Dno,Manager).

office(Dname,Office#,Location,Phone) :-
     department_record(Dname,Dno,Manager),
     office_record(Office#,Location,Phone).

equipment(Dname,Office#,Serial#,Name,Cost) :-
     department_record(Dname,Dno,Manager),
```

```
     office_record(Office#,Location,Phone),
     equipment_record(Serial#,Name,Cost).

project(Dname,Pname,Budget,Startdate) :-
     department_record(Dname,Dno,Manager),
     project_record(Pname,Budget,Startdate).

employee(Ename,Salary,Age,Address) :-
     employee_record(Ename,Salary,Age,Address).

project_employee(Dname,Pname,Ename) :-
     projemp_virtual_record(Dname,Pname,Ename).
```

Figure 6.15 Relational View Definition in Horn Clause

(a) Inclusion Dependencies

```
∀ Dname, ∀ Office#, ∀ Location, ∀ Phone, ∃ Dno, ∃ Manager
(office(Dname,Office#,Location,Phone)
 --> department(Dname,Dno,Manager))

∀ Dname, ∀ Office#, ∀ Serial#, ∀ Name, ∀ Cost,
∃ Dno, ∃ Manager
(equipment(Dname,Office#,Serial#,Name,Cost)
 --> department(Dname,Dno,Manager))

∀ Dname, ∀ Office#, ∀ Serial#, ∀ Location, ∀ Phone,
∃ Dno, ∃ Manager
(equipment(Dname,Office#,Serial#,Name,Cost)
 --> office(Dname,Office#,Location,Phone))

∀ Dname, ∀ Pname, ∀ Ename, ∃ Budget, ∃ Startdate
(project_employee(Dname,Pname,Ename)
 --> project(Dname,Pname,Budget,Startdate))
```

```
∀ Dname, ∀ Pname, ∀ Ename, ∃ Salary, ∀ Age, ∃ Address
(project_employee(Dname,Pname,Ename)
 --> employee(Ename,Salary,Age,Address))

(b) Functional Dependencies

∀ Dname, ∀ Dno1, ∀ Manager1, ∀ Dno2, ∀ Manager2
(department(Dname,Dno1,Manager1)
 department(Dname,Dno2,Manager2)
 --> (Dno1=Dno2, Manager1=Manager2))

∀ Dname, ∀ Office#, ∀ Location1, ∀ Phone1, ∀ Location2,
∀ Phone2
(office(Dname,Office#,Location1,Phone1),
 office(Dname,Office#,Location2,Phone2)
 --> (Location1=Location2, Phone1=Phone2))

∀ Dname, ∀ Office#, ∀ Serial#, ∀ Name1, ∀ Cost1, ∀ Name2,
∀ Cost2
(equipment(Dname,Office#,Serial#,Name1,Cost1)
 equipment(Dname,Office#,Serial#,Name2,Cost2)
 --> (Name1=Name2, Cost1=Cost2))

∀ Dname, ∀ Pname, ∀ Budget1, ∀ Stratdate1, ∀ Budget2,
∀ Startdate2,
(project(Dname,Pname,Budget1,Startdate1)
 project(Dname,Pname,Budget2,Startdate2)
 --> (Budget1=Budget2, Startdate1=Startdate2))

∀ Ename, ∀ Salary1, ∀ Age1, ∀ Address1, ∀ Salary2,
∀ Age2, ∀ Address2
(employee(Ename,Salary1,Age1,Address1)
 employee(Ename,Salary2,Age2,Address2)
 --> (Salary1=Salary2, Age1=Age2, Address1=Address2))
```

```
∀ Dname, ∀ Dno1, ∀ Manager1, ∀ Dno2, ∀ Manager2
(department_record(Dname,Dno1,Manager1)
 department_record(Dname,Dno2,Manager2)
 --> (Dno1=Dno2, Manager1=Manager2))

∀ Office#, ∀ Location1, ∀ Phone1, ∀ Location2, ∀ Phone2
(office_record(Office#,Location1,Phone1),
 office_record(Office#,Location2,Phone2)
 --> (Location1=Location2, Phone1=Phone2))

∀ Serial#, ∀ Name1, ∀ Cost1, ∀ Name2, ∀ Cost2
(equipment_record(Serial#,Name1,Cost1)
 equipment_record(Serial#,Name2,Cost2)
 --> (Name1=Name2, Cost1=Cost2))

∀ Pname, ∀ Budget1, ∀ Stratdate1, ∀ Budget2, ∀ Startdate2,
(project_record(Pname,Budget1,Startdate1)
 project_record(Pname,Budget2,Startdate2)
 --> (Budget1=Budget2, Startdate1=Startdate2))

∀ Ename, ∀ Salary1, ∀ Age1, ∀ Address1, ∀ Salary2,
∀ Age2, ∀ Address2
(employee_record(Ename,Salary1,Age1,Address1)
 employee_record(Ename,Salary2,Age2,Address2)
 --> (Salary1=Salary2, Age1=Age2, Address1=Address2))
```

Figure 6.16 Integrity Constraints in Logic for Schema in Figure 6.15

CHAPTER 7
QUERY PROCESSING

When a high-level query is posed using ESQL over a global schema, it is first transformed to a canonical form, logic query, which is represented in domain relational calculus. This query, suitable for subsequent manipulation, is then simplified using functional, inclusion, or data integration dependencies. One way to simplify the query is to eliminate redundant predicates. Simplified global query (still expressed in relational calculus) is transformed to an algebraic query that is expressed in an executable form. Also, the simplified global query is decomposed into subqueries that are executable in local databases.

In Section 7.1, we present the overall system architecture for query processing and take an example to illustrate how a global query is processed. Section 7.2 shows how to compile a intensional database (IDB) predicate into extensional database (EDB) predicates. Section 7.3 shows how to use functional, inclusion, and data integration dependencies to simplify a global query. In Section 7.4 we decompose a global query into subqueries which we consider as query compilation.

7.1 System Architecture

The system architecture is presented in Figure 7.1. In the following section, we briefly describe the main functional components of this system, and explain their interactions in the process of evaluation of a global query.

7.1.1 Front-End Translation

Assuming that the global user interface uses ESQL, a query written in ESQL is transformed to a logic query, which consists of Horn clauses. The logic query is the intermediate language representation that preserves the intent of a user language while serving as the canonical form that is equivalent but more efficient for query processing. Therefore, the query represented in Horn clause bridges the gap between the way queries are represented in a high-level declarative language, and the way queries are manipulated for optimization and decomposition. In addition, it has the following advantages:

1) It is so general that it can be used as an intermediate language in a variety of different database query languages at the user interface. If a new query language is required for the user interface, the corresponding translator is added without impacting the existing query processing block.

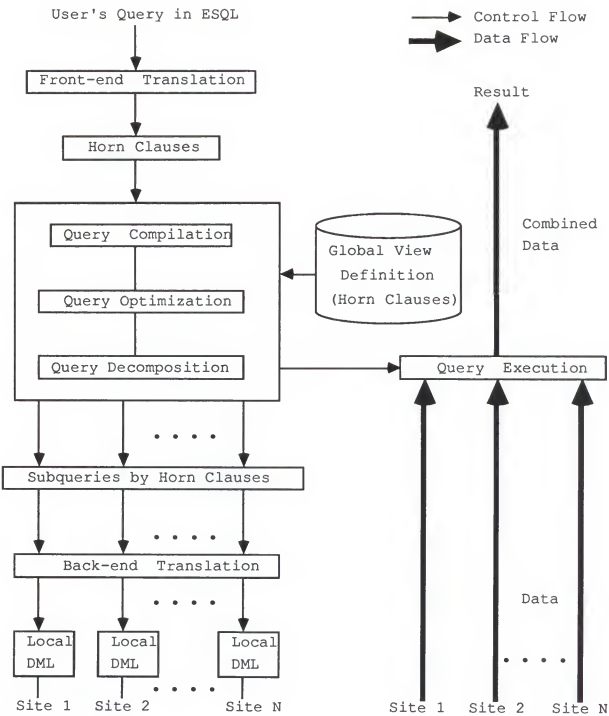

Figure 7.1 Query Processing Architecture

2) The logic programming language itself is used as the database query language. It facilitates development of the query processing module without the need for a high-level user interface.

7.1.2 Query Compilation and Evaluation

The logic query translated from a high-level user interface (e.g, ESQL) is considered to be composed of many levels of rules (i.e., Horn clauses) eventually referencing relations resident in local databases. In other words, local databases are converted to relational view using Horn clauses and component schemas represented in relational view are integrated into a global schema using Horn clauses, and so on. The local databases correspond to extensional databases (EDB), which are actually stored in the local databases. The relations defined by logical rules correspond to intensional databases (IDB). Thus, the global query over the IDB relation is composed of many levels of Horn clauses ultimately corresponding to EDB relations. The evaluation of IDB relation to access EDB relation is done in terms of system-supported unification in logic programming. However, the execution model of current PROLOG is quite different from that of a conventional database system. In PROLOG, the processing of rules is intertwined with the accessing of facts in the sense that the variable bindings between inter-rule

unification are done at run time and the other solutions are produced through backtracking. To avoid the difference in query computation between PROLOG and a conventional DBMS, rules are compiled into sequences of relational algebraic operations on base relations. Hence, variable bindings during inter-rule unification are conducted at compile time and set-oriented processing is performed instead of a tuple-at-a-time processing. Moreover, the compilation plays an important role in decomposition and optimization. The decomposition of global query into subqueries is automatically achieved during compilation. Optimization techniques such as pushing selection as early as possible or distribution of unary operations over union are carried out during compilation without any extra work. After compilation, a query contains only base relations and join and union operations. Finally, an optimized query is executed in an environment in which set-oriented computation is supported.

7.1.3 Back-End Translation

The decomposed subqueries are distributed according to the locations of the corresponding relations. If some subqueries contain relations located in the same location and have a join between them, then the subqueries can be sent as a single query rather than sending separate subqueries. Each subquery is then translated into the corresponding local

database query language.

Example 7.1: Let us consider a simple example that illustrates the processing of a query issued over the global schema given in Figure 5.3: Find the faculty members who are both at the main and branch campuses and earn more than $50,000 a year. The system-generated outputs at each step are as follows.

Query in ESQL

```
SELECT    f.name
FROM      f IS faculty AS
          INTS(main_faculty,branch_faculty)
WHERE     f.salary > 50,000
```

Logic Query translated by Front-end Translator

```
query(Name) :-
     faculty(Name,Salary,_,X),
     X = ints(main,branch),
     Salary > 50,000.
```

The IDB (intensional database) predicate faculty is defined in Chapter 6. If we substitute the IDB rule for faculty, we obtain the following:

Substituting corresponding IDB for faculty

```
query(Name) :-
     main_faculty(Name,Salary1,Office#),
     office(Office#,Main_ph),
```

```
branch_faculty(Name,Salary2,Branch_ph),
Phone = [Main_ph,Branch_ph],
Salary = Salary1 + 12*Salary2
Salary > 50,000.
```

Query Simplification

1) Applying Data Integration Dependency

In the above query, Phone, which is the attribute of FACULTY, is not used in goal and subgoals, so that it can be removed. Also, since Phone determines Main_ph and Branch_ph, the Main_ph and Branch_ph are treated as 'don't care' argument in the corresponding predicate.

```
query(Name) :-
     main_faculty(Name,Salary1,Office#),
     office(Office#,_),
     branch_faculty(Name,Salary2,_),
     Salary = Salary1 + 12*Salary2
     Salary > 50,000.
```

2) Applying Inclusion Dependency

Since the Office# in relation main_faculty is subset of the Office# in relation office and the other argument of office is not used, we can remove the predicate involving office.

```
query(Name) :-
     main_faculty(Name,Salary1,_),
     branch_faculty(Name,Salary2,_),
     Salary = Salary1 + 12*Salary2
     Salary > 50,000.
```

Decomposed Subqueries

```
subquery1(Name,Salary) :-
     main_faculty(Name,Salary,_).
subquery2(Name,Salary) :-
     branch_faculty(Name,Salary,_).
```

Local Query Languages translated by Back-end Translator

```
site1:
     SELECT    Name, Salary
     FROM      main_faculty
site2:
     RANGE OF b IS branch_faculty
     RETRIEVE (b.name, b.salary)
```

7.2 Query Compilation

As we mentioned in Chapter 5, IDB (intensional database) predicates are defined by logical rules. We can eliminate IDB (intensional database) predicates appearing in a query by substituting for each subgoal with predicate p, the body of each of the rules for p, after unifying the rule head with the subgoal. The essential step when substituting an IDB

predicate in a query for its definition is unification. First, we look into the unification process and later illustrate query compilation using this unification. Unification [Chan73, Lloy87] is the operation whereby we take two atomic formulas, each with variables, and find a substitution for each variable, so that the two formulas become identical. Such a substitution is called a unifier. In unification, the expressions we substitute for variables will often themselves involve variables. Thus, any expressions could be substituted for these variables and the resulting formulas would still be identical. As a consequence, there may be more than one unifier for a pair of formulas. Formally, a unifier of two formulas F and G is any substitution θ such that $F\theta = G\theta$; it is their most general unifier (MGU) if and only if, for every other unifier σ of F and G, there exists a substitution γ such that $\sigma = \theta\gamma$ [Chan73, Lloy87]. In other words $F\sigma$ is a substitution instance of $F\theta$. The MGU has the property that any other unifier can be constructed from the MGU by substitution for its variables. Thus, the MGU is the simplest substitution that makes the two formulas identical. To unify an IDB predicate with the rule head, we need to look at the basic unification algorithm. The following algorithm is used in the terms of the formula free from function symbols.

Algorithm 7.1: Unification without Function Symbols

INPUT: Atomic formulas F and G with variables X1,...,Xn and Y1,...,Yn, respectively.

OUTPUT: The unification of F and G or an indication of failure.

METHOD: We use τ for the MGU. Let τ(Xi) and τ(Yi) be the result of unification for each occurrence of Xi in F and Yi in G, respectively. Initially, τ(Xi) and τ(Yi) are undefined for all i. We then apply the following PROCEDURE to F and G. If the call to the PROCEDURE succeeds (returns TRUE), then τ(Xi) and τ(Yi) will be defined for all i, and this substitution is the unification that is produced as output. If the call fails (returns FALSE), then the given atomic formulas F and G are not unifiable.

```
PROCEDURE UnificationWithoutFunctionSymbols(X,Y): boolean;

BEGIN
   IF X is a single variable
      THEN IF τ(X) is undefined
              THEN IF Y is a single variable

                      /* X and Y are variables and unified
                         with Y */
                      THEN BEGIN
                              τ(X) := Y;
                              τ(Y) := Y;
                              RETURN(TRUE);
                           END

                      /* variable X is unified with
                         constant Y */
                      ELSE BEGIN
                              τ(X) := Y;
                              RETURN(TRUE);
                           END
```

```
                ELSE IF τ(X) = Y
                        THEN RETURN(TRUE)

                        /* τ(X) is defined, but not equal to
                           Y */
                        ELSE RETURN(FALSE)
        ELSE IF Y is a single variable
                THEN IF τ(Y) is undefined

                        /* variable Y is unified with
                           constant X */
                        THEN BEGIN
                                τ(Y) := X;
                                RETURN(TRUE);
                             END
                        ELSE IF τ(Y) = X
                                THEN RETURN(TRUE)

                                /* τ(Y) is defined, but not
                                   equal to X */
                                ELSE RETURN(FALSE)

                /* X and Y are both constants */
                ELSE IF X = Y
                        THEN RETURN(TRUE)
                        ELSE RETURN(FALSE)
END
```

For example, two formulas p(X,Y,Z) and p(Y,W,a) are unified with p(Y,Y,a) giving us $\tau(X) = \tau(Y) = Y$, $\tau(Y) = Y = \tau(W)$ and $\tau(Z) = a$. However, p(X,X) cannot be unified with p(a,b) because the algorithm succeeds in the pair of the first arguments X and a, giving us $\tau(X) = a$, but for the second arguments X and b it checks that $\tau(X) = b$, and fails, causing the entire algorithm to fail.

In the query compilation each IDB predicate is unified with the corresponding rule head. If there exists unification, the IDB predicate is replaced with the rule body with variables changed in terms of unification. This process

continues until the body of query consists only of EDB predicates. We assume that there are no recursive definitions.

Algorithm 7.2: Query Compilation

INPUT: A query defined with IDB predicates.

OUTPUT: A compiled query only with EDB predicates.

METHOD: First, if there exists a rule R corresponding to an ordinary predicate p in a query Q, then the variables in the rules are renamed such that R shares no variables with the query Q. Second, we find the MGU of the head of R and the predicate p so that the predicate p can be replaced with the body of R. Third, the variables of the query and the rule are changed in accordance with the result of unification. Now we can obtain a new query by replacing the predicate p with the body of the rule R. This process continues recursively until all the IDB predicates in the query are replaced with EDB predicates.

```
PROCEDURE QueryCompilation(Q);

BEGIN
   FOR each occurrence of ordinary predicate p in a subgoal
         G of a given query Q DO
      BEGIN FOR each rule R with head predicate p DO
               BEGIN
                  rename the variables of R so that R shares
                     no variable with Q;

                  perform the unification of the head of R
                     with the subgoal G and let τ be the
```

```
                    most general unifier of the head of R
                    and the subgoal G;

                 change the variables in the query Q and
                    rule R to τ(Q) and τ(R) in accordance
                    with the result of the MGU τ;

                 create a new query Q' by taking τ(Q)
                    and replacing the subgoal τ(G) by the
                    body of τ(R);

                 QueryCompilation(Q');
              END;
      END;
   output(Q');
END
```

Example 7.2: Suppose p has two rules R1 and R2, and the query, Q, is given as follows:

```
R1: p(X,Y,Z) :- s(X,Y), t(Y,Z,a).
R2: p(X,Y,Z) :- u(X,Y,Z), v(X,b,c).
Q: q(X,Y) :- p(X,Z,d), w(Z,Y).
```

We begin by rewriting R1 and R2 so that they share no variables with Q; they become

```
R1: p(V1,V2,V3) :- s(V1,V2), t(V2,V3,a).
R2: p(V1,V2,V3) :- u(V1,V2,V3), v(V1,b,c).
```

Now, we unify the head of R1, p(V1,V2,V3), with the subgoal p(X,Z,d), which gives us the MGU $\tau(X) = \tau(V1) = V1$, $\tau(Z) = \tau(V2) = V2$, and $\tau(V3) = d$. For Y, which is not involved in the unification, we can assume $\tau(Y) = Y$. Then Q becomes

```
τ(Q): q(V1,Y) :- p(V1,V2,d), w(V2,Y).
```

and τ(R1) is

τ(R1): p(V1,V2,d) :- s(V1,V2), t(V2,d,a).

When we substitute the body of τ(R1) for the subgoal p(V1,V2,d) in τ(Q), we obtain the new query

Q1: q(V1,Y) :- s(V1,V2), t(V2,d,a), w(V2,Y).

Also, when we unify the head of R2 with the subgoal p(X,Z,d) of Q, we get the MGU σ(X) = σ(V1) = V1, σ(Z) = σ(V2) = V2, and σ(V3) = d. For Y, which is not involved in the unification, we can assume σ(Y) = Y. Then, substituting the body of (R2) for the p-subgoal in (Q), we get another new query

Q2: q(V1,Y) :- u(V1,V2,d), v(V1,b,c), w(V2,Y).

7.3 Query Simplification

We simplify a query using integrity constraints such as functional, inclusion and data integration dependencies. We consider each type in turn.

1) Functional Dependency

Let R(A1,...,An) be a relation schema, and let X and Y be subsets of {A1,...,An}. The functional dependency (FD), denoted by X -> Y, states that "X functionally determines Y" or "Y functionally depends on X." It specifies a constraint

on the possible tuples that can form a relation instance r of R. The constraints state that for any two tuples t1 and t2 in r such that t1[X] = t2[X], we must also have t1[Y] = t2[Y]. The set of attributes X is called left-hand side (LHS) of the functional dependency and Y is called the right-hand side (RHS). The algorithm shown below is used to simplify a query when applying functional dependencies to a query.

Algorithm 7.3: Applying Functional Dependencies to a Query and Eliminating a Duplicate Predicate.

INPUT: Functional dependencies of relations and a query.
OUTPUT: A query that a duplicate predicate is eliminated.
METHOD: If there exist the same ordinary predicates p's in a query and the LHSs of functional dependency are the same, then the following is conducted on the query. First, we perform the unification of the two same predicates p's and find the MGU for the two p's. Second, the variables in the query Q are changed in accordance with the result of the MGU. Third, a new query is created by removing one of the same predicates p's. This process continues until no more functional dependency is applicable to a query.

```
PROCEDURE ApplyingFunctionalDependency(Q);
BEGIN
   FOR each occurrence of ordinary predicate p in a given
         query Q DO
      IF there is the same predicate as p in Q
```

```
        THEN IF the two same predicates p's have the
                same LHS of functional dependency and
                LHS represents a key

             THEN BEGIN
                    perform the unification of the two
                      p's and let τ be the most general
                      unifier of the two p's;

                    change the variables in Q in
                      accordance with the result of the
                      unification τ;

                    create a new query Q' by removing
                       one of p's;

                    ApplyingFunctionalDependency(Q');
                  END;
END
```

Example 7.3: Let the IDB for a high-paid faculty be defined as follows:

```
high-paid-faculty(Name) :-
     faculty(Name,Salary,Phone), Salary > 50,000.
```

For a query such as "Find the name and phone number of high-paid-faculties,"

```
query(Name,Phone) :-
     high_paid_faculty(Name),
     faculty(Name,Salary,Phone).
```

The following query is generated when an IDB predicate, high_paid_faculty, is replaced by the rule body as shown in Section 7.2.

```
query(Name,Phone) :-
     faculty(Name,Salary,Phone),
```

```
    faculty(Name,Salary1,Phone1),
    Salary1 > 50,000.
```

If we apply the functional dependency (i.e., Name -> Salary,Phone) to this query, we get the MGU τ(Salary) = τ(Salary1) = Salary, τ(Phone) = τ(Phone1) = Phone. When variables are changed in accordance with the result of the MGU and the one of predicate faculty is removed, the query becomes

```
query(Name,Phone) :-
    faculty(Name,Salary,Phone),
    Salary > 50,000.
```

2) Inclusion Dependency

An inclusion dependency for two relations R and S is a statement of the form R[X] $\subseteq$ S[Y], where X and Y are sequences of attributes of R and S respectively, such that X = Y. If Y is a key of S, the inclusion dependency is called key-based. If X consists of nonprime attributes of R in key-based inclusion dependency, then it represents referential integrity constraints. The idea of using referential integrity constraints in query simplification is that when we join the two relations S and R in terms of the key of S and the foreign key of R, if other attributes than the key of the relation S is not used in the query, we can remove S because S is superset of R on join attribute.

Algorithm 7.4: Applying an Inclusion Dependency to a Query and Eliminating a Redundant Predicate

INPUT: Inclusion dependencies of relations and a query.

OUTPUT: A query where redundant predicate is eliminated.

METHOD: If there exists inclusion dependency between the two relations P and S such that P[X] ⊆ S[X], and other arguments (attributes) except X is not used in the query, then the predicate s is removed.

```
PROCEDURE ApplyingInclusionDependency(Q);

BEGIN
   FOR each occurrence of ordinary predicate p DO
      IF there exists a predicate s such that inclusion
           dependency holds between p and s
           (i.e., P[X] ⊆ S[X])
         THEN IF other arguments than X of the predicate s
                   are not used in the query Q
                THEN BEGIN
                        create a new query Q' by removing
                          the predicate s;

                        ApplyingInclusionDependency(Q');
                     END;
END
```

Example 7.4: As an example for using inclusion dependency, consider the IDB defined as follows:

```
faculty(Name,Salary,Phone) :-
     main_faculty(Name,Salary,Office#),
     office(Office#,Phone).
```

We are concerned with phone number of faculty. Now we pose a query over the faculty such as "Find the faculty whose salary

is over $60,000." as follows:

```
query(Name) :-
     faculty(Name,Salary,Phone),
     Salary > 60,000.
```

After substituting IDB definition for IDB predicate, we obtain the query as follows:

```
query(Name) :-
     main_faculty(Name,Salary,Office#),
     office(Office#,Phone),
     Salary > 60,000.
```

Since OFFICE[Office#] ⊇ MAIN_FACULTY[Office#] and Phone is not used in the query, we can remove the office predicate. The query becomes as follow:

```
query(Name) :-
     main_faculty(Name,Salary,Office#),
     Salary > 60,000.
```

3) Data Integration Dependency

The data integration dependency, denoted by X1, ..., Xn -> X, states that the attribute X in the global schema is built from the attributes X1,...Xn in the component schemas. For example, the attribute Salary in the global view relation FACULTY is composed of Salary1 in MAIN_FACULTY and Salary2 in

BRANCH_FACULTY. The idea of using data integration dependency in query simplification is that if the attribute X is not used in the query, then we do not need to compute X from X1,...,Xn as well as to project out the attributes X1,...,Xn.

Algorithm 7.5: Applying Data Integration Dependencies

INPUT: Data Integration Dependencies of relations and a query.
OUTPUT: A query where unused predicates are removed.
METHOD: If the built-in predicate is not used in the query, then we can remove the built-in predicate. In addition, we do not need to project out the arguments corresponding to left-hand side of data integration dependency.

```
PROCEDURE ApplyingDataIntegrationDependency(Q);

BEGIN
   FOR each occurrence of built-in predicate p DO
      IF p is not used in the query
         THEN BEGIN
                 create a new predicate Q' by removing
                    the predicate p;

                 mark don't care to the arguments
                    associated with data integration
                    dependency on p;

                 ApplyingDataIntegrationDependency(Q');
              END;
END
```

7.4 Query Decomposition

When a global query is posed over global schema, it is compiled into a query composed only with EDB predicates. We simplify this query using functional, inclusion and data integration constraints as shown above. Now this global query is decomposed into several subqueries each of which consists of local relations. We shall now describe how to decompose a global query into several subqueries.

Algorithm 7.6 Query Decomposition

INPUT: A query and location information of predicates used in the query.

OUTPUT: Several subqueries decomposed from the given query.

METHOD: Let a global query be given as follows:

Q :- P1,...,Pn,B1,...,Bm.

Where the P's are ordinary predicates and B's are built-in predicates. The ordinary predicates P's are collected according to the location information of each predicate. In case of built-in predicates B's, if the arguments of a built-in predicate is composed of those that belong to the same location, that predicate can be computed in that corresponding location, so that the predicate is collected according to its location.

```
PROCEDURE QueryDecomposition(Q);

BEGIN
   FOR each occurrence of ordinary predicate DO
      collect the predicate according to its location
         information;

   FOR each occurrence of built-in predicate DO
      IF the arguments of the built-in predicate belong to
            the same location
         THEN collect the built-in predicate according to
                  its location information;

   generate each subquery from the predicates that belong to
      the same location;
END
```

In this chapter, we presented the overall system architecture for query processing and gave an example to illustrate how a global query is processed. When we evaluate a global query posed using ESQL, we first translate it into a form of Horn clause representation, which provides a canonical form suitable for subsequent manipulation such as query optimization and decomposition. We showed how to compile a global query into EDB (extensional database) predicates, how to use functional, inclusion, and data integration dependencies to simplify a global query, and how to decompose a global query into subqueries.

CHAPTER 8
CONCLUSION

8.1 Summary and Contributions

In this dissertation we have presented a logic-based approach to address integration and implementation issues of federated databases. The results and contributions of this work can be grouped into four categories as summarized below.

First, in this dissertation we used first-order logic as a language for expressing integration of component schemas. Use of pure relational model does not allow the conceptual integration of nondisjoint data from distinct participating databases. Although relational views can define virtual relations, they are not powerful enough to deal with integration aspects of data such as generalization. The derivation for the global schema from component schemas are represented in the form of rules, i.e., Horn clauses. The integrity constraints such as functional, inclusion, and data integration constraints are also represented in Horn clauses and used in query optimization. Before integration, all local schemas must be converted to a common data model to facilitate schema integration. The conversion from nonrelational schemas (i.e., network and hierarchical) to relational schema and

expressing integrity constraints associated with them have been done using Horn clauses. Thus, using logic as a unified, canonical representation allowed us to facilitate schema integration, global to local query mapping, and application of various optimization strategies in a single framework.

Second, in order to generate a single integrated schema from component relations, the equivalences between two relations, R1 and R2, should be identified in one of five ways: R1 equals R2, R1 includes R2, R1 overlaps R2, R1 contains R2, R1 is_contained_in R2, and R1 is_disjoint_but_integrable_with R2. An algorithm is developed to aid the designer to derive new equivalences from partially known equivalences as well as to check consistency of equivalences that are previously specified.

Third, when global schema is generated based on the five types of equivalences between a pair of relations, we need operations to retrieve occurrences of global view relation associated with its component relations. We have developed an extended form of the SQL language (ESQL) to provide set operations on the component relations. Although conventional SQL allows set operations on the relations that are union compatible, it requires several subqueries and procedural specifications. However, the set operations in ESQL are expressed nonprocedurally in a single statement.

Finally, to demonstrate the viability of logic-based approach, implementation is carried out using KB-PROLOG

[Bocc89], which is a PROLOG database system that provides extended PROLOG functions and supports relational operations augmented with the capability to handle secondary storage. The KB-PROLOG environment that runs on the SUN workstation provides a set of development tools as well as full computational power. Since it is used as an internal DBMS to process the intermediate results based on set-oriented operations, the computation of combined data is done in this environment without any need to access other DBMSs.

8.2 Future Work

For further research and development we would to like to address the following:

First, it is likely that all systems in the federation do not support the same set of operations. In order to ensure that global requests are properly computed, the global query evaluator needs to compensate for functionality that may be lacking in some systems. This is better done with a system that is more powerful than the ones participating in the federation. For example, as opposed to conventional data manipulation languages, logical rules have the power to compute transitive closures, such as managerial hierarchies. In this respect, we think that the use of a deductive database is ideal as it can support the relational model easily and at the same time provide for capabilities that are beyond

relational systems.

Second, one of the extensions beyond this work is to create federated information systems that include not only database systems but also application programs, knowledge base systems or expert systems. Knowledge formulated in some form of logic in knowledge based systems and rule based language used in expert systems can be mapped to first-order logic because their underlying formalisms are the same. Besides, since first-order logic provides formalism in which various statements can be expressed, the knowledge represented in semantic networks or frames can be mapped into first-order logic as shown in Deliyanni and Kowalski [Deli79]. However, the reverse direction mapping is not always possible, because first-order logic is more expressive than structure-oriented representation in the sense that the notions of quantification (either universal or existential) and negation are not easy to represent unambiguously in network and structured data form. However, much work remains to be done in this direction.

REFERENCES

[Banc86] Banchilon, F., and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies," In Proceedings of the ACM SIGMOD International Conference on the Management of Data, pp. 16-52, Washington, D.C., May 1986.

[Bati84] Batini, C., M. Lenzerini, "A Methodology for Data Schema Integration in the Entity-Relationship Model," IEEE Transactions on Software Engineering, Vol.10, No.6, pp. 650-663, November 1984.

[Bati86] Batini, C., M. Lenzerini, and S.B. Navathe, "A Comparative Analysis of Methodologies for Database Schema Integration," ACM Computing Surveys, Vol.18, No.4, pp. 323-364, December 1986.

[Bisk86] Biskup, J., and B. Convent, "A Formal View Integration Method," In Proceedings of the ACM SIGMOD International Conference on the Management of Data, pp. 398-407, Washington, D.C., May 1986.

[Bocc89] Bocca, J., M. Dahmen, G. Macartney, and P.Pearson, "KB-Prolog User Guide," European Computer-Industry Research Center, Munich, September 1989.

[Brei86] Breitbart, Y., P.L. Olson., and G.R. Thompson, "Database Integration in a Distributed Heterogeneous Database System," In Proceedings of the 2nd International Conference on Data Engineering, pp. 301-310, Los Angeles, California, February 1986.

[Chak90] Chakravarthy, U.S., J. Grant, and J. Minker, "Logic-Based Approach to Semantic Query Optimization," ACM Transactions on Database Systems, Vol.15, No.2, pp. 162-207, June 1990.

[Chan73] Chang, C.L., and R.C. Lee, Symbolic Logic and Mechanical Theorem Proving, Academic Press, New York, New York, 1973.

[Chim87] Chimenti, D., A. O'Hare, R. Krishnamurthy, S.Naqvi, S.Tsur, C. West, and C. Zaniolo, "An Overview of the LDL System," IEEE Data Engineering, Vol.10, No.4, pp. 52-62, December 1987.

[Chun90] Chung, C.W. "DATAPLEX: A Access to Heterogeneous Distributed Databases," Communications of the ACM, Vol.33, No.1, pp. 70-80, January 1990.

[Czej87] Czejdo, B., M. Rusinkiewicz, and D.W. Embley, "An Approach to Schema Integration and Query Formulation in Federated Database Systems," In Proceedings of the 3rd International Conference on Data Engineering, pp. 477-484, Los Angeles, California, February 1987.

[Daya84] Dayal, U., and H. Hwang, "View Definition and Generalization for Database Integration in Multibase: A System for Heterogeneous Distributed Databases," IEEE Transactions on Software Engineering, Vol.10, No.6, pp. 628-645, November 1984.

[Deli79] Deliyanni, and R. Kowalski, "Logic and Semantic Networks," Communications of the ACM, Vol.22, No.3, pp. 184-192, March 1979.

[Effe84] Effelsberg, W., and M.V. Mannino, "Attribute Equivalence in Global Schema Design for Heterogeneous Distributed Databases," Information Systems, Vol.9, No.3/4. pp. 237-240, 1984.

[Elma84] Elmasri, R., and S.B. Navathe, "Object Integration in Logical Database Design," In Proceedings of the 1st International Conference on Data Engineering, pp. 418-425, Los Angeles, California, April 1984.

[Elma86] Elmasri, R., J. Larson, and S.B. Navathe, "Schema Integration Algorithms for Federated Databases and Logical Database Design," Tech. Rep. No. CSC-86-9: 8212, Honeywell Corporate Systems Development Division, Camden, Minnesota, 1986.

[Elma89] Elmasri, R., and S.B. Navathe, Fundamentals of Database Systems, The Benjamin/Cummings Publishing Company, Inc, Redwood City, California, 1989.

[Ferr83] Ferrier, A., and C. Stangret, "Heterogeneity in the Distributed Database Management System Sirius-Delta," In Proceedings of the 9th International Conference on Very Large Data Bases, pp. 45-53, Florence, Italy, October/November 1983.

[Gall84] Gallaire, H., J. Minker, and J.M. Nicolas, "Logic and Databases: A Deductive Approach," ACM Computing Surveys, Vol.16, No.2, pp. 153-185, June 1984.

[Hamm79] Hammer, M., and D. McLeod, "On Database Management System Architecture," Tech. Rep. MIT/LCS/TM-141, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1979.

[Heim85] Heimbigner D., and Mcleod D., "A Federated Architecture for Information Management," ACM Transactions on Office Information Systems, Vol.3, No.3, pp. 253-278, July 1985.

[Howe87] Howels,D.I., N.J.Fidden, W.A.Gray, "A Source-so-Source Meta Translation System for Relational Query Languages," In Proceedings of the 13th International Conference on Very Large Data Bases, pp. 227-234, Brighton, England, September 1987.

[Kaul90] Kaul, M., K. Drosten, and E.J. Neuhold, "ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views," In Proceedings of the 6th International Conference on Data Engineering, pp. 2-10, Los Angeles, California, February 1990.

[Kim82] Kim, W. "On Optimizing an SQL-like Nested Query," ACM Transactions of Database Systems, Vol.7, No.3, pp. 443-469, September 1982.

[Kris87] Krishnamurthy, V., S.Y.W. Su, H. Lam, M. Mitchell, and E. Barkmeyer, "A Distributed Database Architecture for an Integrated Manufacturing Facility," Proc. of the International Conference on Data and Knowledge Systems for Manufacturing and Engineering," Computer Society Press of the IEEE, pp. 4-13, October 1987.

[Land82] Landers, T., and R. Rosenberg, "An Overview of Multibase," In Distributed Databases, H-J. Schneider (ed.), North-Holland, pp. 153-184, New York, New York, 1982.

[Lars89] Larson, P., S.B. Navathe, and R. Elmasri, "A Theory of Attribute Equivalence in Databases with Application to Schema Integration," IEEE Transactions on Software Engineering, Vol.15. No.4, pp. 449-463, April 1989.

[Lind89] Linda, G. D., "Performing Operations over Mismatched Domains," In Proceedings of the 5th International Conference on Data Engineering, pp. 572-581, Los Angeles, California, February 1989.

[Litw86] Litwin, W., and A. Abdellatif, "Multidatabase Interoperability," IEEE Computer, Vol.19, No.12, December 1986.

[Litw90] Litwin, W., L. Mark, and N. Roussopoulos, "Interoperability of Multiple Autonomous Databases," ACM Computing Surveys, Vol.22, No.3, pp. 267-293, September 1990.

[Lloy87] Lloyd, J.W., Foundations of Logic Programming, Second Edition, Springer-Verlag, New York, New York, 1987.

[Mann84] Mannino, M.V, and W. Effelsberg, "Matching Techniques in Global Schema Design," In Proceedings of the 1st International Conference on Data Engineering, pp. 418-425, Los Angeles, California, April 1984.

[Motr87] Motro, A., "Superviews: A Virtual Integration of Multiple Databases," IEEE Transactions on Software Eng., Vol.13, No.7., pp. 785-798, July 1987.

[Motr81] Motro, A., P. Buneman, "Constructing Superviews," In Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 54-64, Ann Arbor, Michigan, April/May 1981.

[Nava80] Navathe, S.B., "An Intuitive Approach to Normalize Network Structured Data," In Proceedings of the 6th International Conference on Very Large Data Bases, pp. 350-358, Montreal, Canada, October 1980.

[Nava82] Navathe, S.B., and S.G. Gadgil, "A Methodology for View Integration in Logical Database Design," In Proceedings of the 8th International Conference on Very Large Data Bases, pp. 142-164, Mexico City, Mexico, September 1982.

[Nava86] Navathe, S.B., R. Elmasri, and J.A. Larson., "Integrating User Views in Database Design," IEEE Computer, Vol.19, No.1, pp. 50-62, January 1986.

[Nava89] Navathe, S.B., S.K. Gala, and S. Geum, "A Federated Approach to Loose-Coupled Integration of Multiple Information Systems," Technical Report, Database Systems R & D Center, University of Florida, January 1989.

[Rusi88] Rusinkiewicz, M., B. Czejdo, R. Elmasri, D. Georakopoulos, A, Jamoussi, G. Karabatis, Y. Li, L.Loa, J. Gilbert, and R. Musgrove, "Query Processing in OMNIBASE - A Loosely Coupled Multibase System," Technical Report UH-CS-88-05, University of Houston, February 1988.

[Sava91] Savasere, A., A.P. Sheth, S. Gala, S.B. Navathe, and H. Marcus, "On Applying Classification to Schema Integration," In the First International Workshop on Interoperability in Mutidatabase Systems, pp. 258-261, Kyoto, Japan, April 1991.

[Shet88] Sheth, A.P., J.A. Larson, J. Cornellio, and S.B. Navathe, "A Tool for Integrating Conceptual Schemas and User Views," In Proceedings of the 4th International Conference on Data Engineering, pp. 176-183, Los Angeles, California, February 1988.

[Shet90] Sheth, A.P., and J.A. Larson, "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases," ACM Computing Surveys, Vol. 22, No. 3, pp. 183-236, September 1990.

[Ship81] Shipman, D., "The Functional Data Model Data Language DAPLEX," ACM Transactions on Database Systems. Vol.6, No.1, pp. 140-173, March 1981

[Temp87] Templeton, M., D. Brill, S.K. Dao, E. Lund, P. Ward, A.Chen, and R. MacGregor. "Mermaid - A Front-end to Distributed Heterogeneous Databases," In Proceedings of the IEEE, Vol.75, No.5, pp. 695-708, May 1987.

[Thom90] Thomas, G., G.R. Thompson, C.W. Chung, E. Barkmeyer, F. Carter, M. Templeton, S. Fox, and B. Hartman, "Heterogeneous Distributed Database Systems for Production Use," ACM Computing Surveys, Vol.22, No.3, pp. 237-266, September 1990.

[Ullm85] Ullman, J., "Implementation of Logical Query Languages for Databases," ACM Transactions on Database Systems, Vol.10, No.3, pp. 289-321, September 1985.

[Ullm88] Ullman, J., Principles of Database and Knowledge-Base Systems, Vol.1 & II, Computer Science Press, Rockville, Maryland, 1988.

[Zani79] Zaniolo, C., "Design of Relational Views over Network Schemas," In Proceedings of the ACM SIGMOD International Conference on Management Data, pp. 179-190, Boston, Massachusetts, May/June 1979.

[Zani83] Zaniolo, C., "The Database Language GEM," In Proceedings of the ACM SIGMOD International Conference on Management Data, pp. 207-218, San Jose, California, May 1983.

BIOGRAPHICAL SKETCH

The author was born on May 26, 1952, in Chuncheon, Korea. He received the degree of Bachelor of Science from Seoul National University, Seoul, Korea, in February 1976, and majored in electronics engineering in industrial education department. He received the degree of Master of Science in electrical engineering from the University of Florida in December 1987, and will receive the degree of Doctor of Philosophy in electrical engineering in May 1992.

Following graduation from undergraduate school, he worked at the Agency for Defense Development as a researcher for six years participating in digital circuit design and microprocessor-based applications for electronic equipments. Before coming to the University of Florida in 1985, he was employed at the industrial company, Handok, involving microprocessor-based systems development and later at Data Communication Corp., working as a computer systems engineer.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Shamkant B. Navathe, Chairman
Professor of Electrical Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Herman Lam, Co-Chairman
Associate Professor of Electrical Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

John Staudhammer
Professor of Electrical Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Sharma Chakravarthy
Associate Professor of Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Manuel Bermudez
Associate Professor of Computer and Information Sciences

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

May 1992

Winfred M. Phillips
Dean, College of Engineering

Madelyn M. Lockhart
Dean, Graduate School